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

Ezequiel Santamaría Navarro

25 de junio 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 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.
- 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.

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) [2].

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 [3]. 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

2.3. 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[1].

Acceso: Página web oficial de codemirror.

Capítulo 3

Análisis de objetivos y metodología

3.1. Desarrollo de un entorno de programación

3.1.1. Compilador del lenguaje ZL

3.1.2. Editor de texto inteligente

3.2. Definición del lenguaje ZL

3.2.1. Descripción de la notación BNF ampliada

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.

Por otro lado, los número en el lenguaje permiten indicar base 2, 10 o 16 de la siguiente manera:

- 51966 como número en base 10
- 51966|10 como el mismo número
- $CAFE$|16 o $cafe$|16 como el mismo número en base hexadecimal.
- 1100101011111110|2 como el número en base binaria.

3.2.3. Descripción de la semántica

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

4.2. 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 manual*, `https://codemirror.net/doc/manual.html#overview`, 2016.

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

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

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