La Torre: Construye tu propia aventura en InformATE!

Contenidos

La Torre: Construye tu propia aventura con InformATE

Zak McKraken
(basado en el cursillo de NMP de Carlos Sánchez)

13 de junio de 2002

Parte I: Lo más básico

Este cursillo trata de presentar los conceptos de programación de Inform de una forma guiada por un objetivo: crear una aventura dada. He tomado como aventura la propuesta por Carlos Sánchez en su serie de tutoriales para NMP

A la vez, se irán mostrando las diferencias de enfoque entre los métodos clásicos de creación de aventuras (los llamados "tipo PAWS", como eran SINTAC, NMP, SKC, PAWS, etc...) y los modernos métodos "orientados a objetos" (como el propio Inform).

Debido a la naturaleza introductoria del cursillo, y debido también al enfoque comparativo con PAWS, las explicaciones son muy largas. No es un cursillo de "manos a la obra y en dos minutos tengo el juego", sino que es más bien de explicación detallada de todos los conceptos. Antes de aprender a correr hay que aprender a caminar. Esta lentitud inicial del aprendizaje servirá para que puedas ganar velocidad por tí mismo.

La aventura que nos servirá de ejemplo es muy sencilla: un sólo objetivo, escapar de una torre en la que hemos sido encerrados.

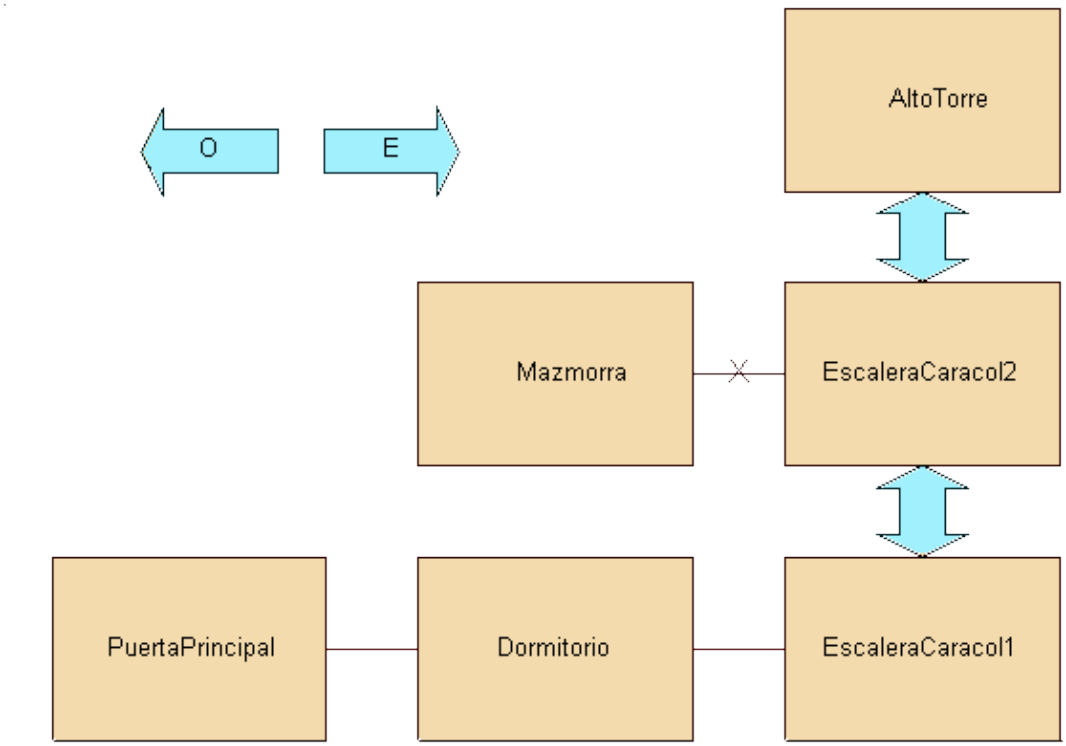
Por dónde empezar.

Evidentemente, antes de nada, tienes que asegurarte de disponer de todos los ingredientes necesarios para la programación con Inform. Asegurate de tener la última versión de todo, para poder seguir este cursillo. Si no sabes lo que necesitas, mira este enlace. Incluso si lo sabes, se recomienda mirarlo, pues se explica además cómo se usan estos ingredientes para "compilar" un juego, algo que tendrás que hacer muy a menudo a lo largo de este cursillo.

Ahora podemos empezar a diseñar el juego. Lo primero, evidentemente, es el guión, que puede ser una idea genérica que se vaya concretando a medida que se programa, o puede ser una historia totalmente detallada, incluyendo incluso los rompecabezas del juego. Cada uno tiene su estilo a la hora de diseñar el guión, pero en este cursillo te lo vamos a dar ya diseñado y con bastante detalle, para que sepas lo que te espera.

El guión del juego

Estás encerrado en una torre, de la que tienes que escapar. El mapa de la torre es el siguiente, las flechas dobles quieren decir "subir o bajar".



El juego comienza en la habitación llamada `PuertaPrincipal`, pero no se puede salir de la torre desde ahi. La salida de la torre está en la `Mazmorra`. Para salir, el jugador deberá:

- Descubrir la entrada a la mazmorra (que es secreta y se revela sólo empujando una antorcha que hay en la escalera)
- Encontrar un cuchillo registrando el esqueleto que hay en la mazmorra.
- Examinar las camas del `Dormitorio`, para descubrir unas fundas atadas con correas
- Cortar las correas con el cuchillo y coger la funda.
- Examinar la ventana de la mazmorra para descubrir un barrote suelto, el cual debe quitar
- Atar la funda de la cama a otro barrote de la ventana.
- Salir por la ventana de la mazmorra.

Antes de lanzarnos a programar todo esto, debemos asentar claramente los principios de programación bajo Inform, que son bien distintos de los de PAWS, por esto iremos muy, muy despacio. No te impacientes, ¡todo llegará!

El juego InformATE más pequeño del mundo

Lo siguiente es el esquema básico que debe seguir todo juego Inform. Más adelante iremos rellenando el esquema. Abre tu editor de texto y copia esto:

```
Constant Historia "La Torre";
Constant Titular "^(c) 2002 Zak y Carlos^";
Include "EParser";
! Aqui iría la definición de mensajes de librería.

Include "Acciones";
! Aqui irian las definiciones de las localidades
! y los objetos del juego
```

```
[ Inicializar ;
];

Include "Gramatica";
! Aqui irían los verbos y acciones específicas de nuestro juego.
```

Copialo todo tal cual, especialmente pon cuidado en los punto y coma. Si no quieres copiarlo, tienes una versión ya tecleada aqui. Aunque parezca increible, eso ya es un programa completo en Inform, puedes compilarlo y tratar de jugarlo (tienes la versión lista para ser jugada en el enlace anterior). Naturalmente no podrás hacer gran cosa, ya que no hay localidades ni objetos. Aún así, puedes intentar una gran variedad de acciones (intenta saltar, cantar, ir al norte o al sur, e incluso SAVE y LOAD). Verás que el juego tiene respuestas para todo ¿de dónde las saca? ¡No hemos programado ninguna de esas respuestas!

He aquí uno de los principios básicos de Inform que lo diferencia claramente de PAWS. En PAWS, el programador tenía que programar las respuestas a cada posible verbo bajo cada posible circunstancia. Los casos que no eran contemplados por el programador, no existían en el juego. En Inform, en cambio, el programador sólo necesita especificar las respuestas a algunos verbos y en algunos casos. Todos los casos que el programador no contemple, serán manejados de forma automática por la librería. Es la librería la que tiene todas esas respuestas preparadas para las acciones más corrientes que el jugador intentará.

Pasemos a explicar línea a línea el programa que has copiado. Puedes saltarte esto e ir directamente a la programación del mapa si ya estás familiarizado con la sintaxis de Inform.

```
Constant Historia "La Torre";
```

Con esto definimos una constante llamada Historia. Esta constante debe definirse en todo juego Inform y contiene el título del juego. El valor de esta constante tiene que ser un texto entre comillas. En nuestro caso el texto es `"La Torre"` Observa que al final del todo debe ponerse un punto y coma. En Inform, todas las "declaraciones" van terminadas por punto y coma.

```
Constant Titular "^(c) 2002 Zak y Carlos^";
```

Este es análogo al anterior, sólo que ahora la constante que definimos se llama `Titular` y su valor es `"^(c) 2002 Zak y Carlos^"`. La constante `Titular` debe existir también en todo juego Inform y es el "subtítulo" del juego. El signo ^ representa un salto de línea, es decir, causa que antes de imprimir el texto el cursor pase a la línea siguiente (y otro salto tras escribir el texto, puesto que hay otro ^ al final).

```
Include "EParser";
```

`Include` es lo que se llama una "directiva", esto es, una orden que le das a Inform para que haga algo. En este caso lo que le estás ordenando es que coja el fichero llamado `EParser.h` (la `h` se la pone automáticamente) y lo inserte en este punto. El resultado es el mismo que si hubieras copiado a mano tú mismo todo el contenido de `EParser.h` en este punto del fichero, sólo que usando `Include` el código fuente del juego queda mucho más limpio y legible (¡el fichero `EParser.h` son miles de líneas!)

El fichero `EParser.h` contiene el "parser en Español" de Inform. Es una parte de la librería que se ocupa de leer lo que el jugador escribe, tratar de comprenderlo y generar acciones dentro del juego. Enseguida veremos qué es eso de las acciones.

```
! Aqui iría la definición de mensajes de librería.
```

Esta línea es lo que se llama un "comentario". Si Inform encuentra un signo de exclamación (!) que no esté formando parte de un texto entrecomillado, automáticamente ignorará todo lo que sigue en esa misma línea. Esto lo usaremos para poner anotaciones en el código. Estas anotaciones nunca formarán parte del juego final, una vez compilado, puesto que Inform las ignora.

En este caso la anotación es un recordatorio de dónde tendríamos que poner nuestras definiciones de mensajes de librería, si quisieramos ponerlas. En este juego no las pondremos de momento. Las definiciones de mensajes de librería son un poco como la tabla de mensajes de sistema de PAWS. Es donde se definen textos como "No puedes ir por ahi", "Cogido", "Cantas fatal", etc... Es decir, las respuestas por defecto de la librería ante cada posible acción. Si no definimos nosotros unas, la librería proporcionará otras ya preparadas. Puedes cambiar algunas de ellas si no te gustan sin necesidad de redefinirlas todas (y sin tener que modificar la librería). Luego veremos cómo.

```
Include "Acciones";
```

De nuevo mediante `Include`, cargamos en nuestro juego el fichero `Acciones.h`, que contiene el código mediante el cual la librería sabe qué es lo que debe hacer ante cada posible acción. (Enseguida veremos qué son las acciones, son un concepto clave de Inform)

```
! Aqui irian las definiciones de las localidades
! y los objetos del juego
```

Ahora se trata de unos comentarios recordándonos dónde irán los objetos y las localidades. Por el momento no hay ni objetos ni localidades.

```
[ Inicializar ;
];
```

Esto es una rutina "libre". En Inform hay dos tipos de rutinas, las rutinas "libres" y las rutinas "incrustadas en objetos". Las libres son las que aparecen, como esta, en medio del código, las incrustadas en objetos aparecen siempre dentro de un objeto, luego daremos más detalles.

Una rutina no es más que un conjunto de instrucciones a las que hemos decidido agrupar y dar un nombre. En cierta manera se parece al concepto de "proceso" del PAWS, solo que en PAWS los procesos se identificaban mediante números (proceso 6, proceso 7,...) y en inform se identifican mediante nombres (rutina Inicializar, rutina LugarNuevo, ...)

Todas las rutinas libres tienen la misma estructura:

```
[ Nombre_de_la_rutina v1 v2 v3;
  instruccion1;
  instruccion2;
  ...
];
```

Es decir:

- El signo `[` indica que comienza la definición de una rutina "libre".
- A continuación va el nombre de la rutina, que es cualquier combinación de letras, números y el signo _ (no se pueden poner acentos ni eñes)
- Seguidamente, separados por espacios, los nombres de las "variables locales" de la rutina. El concepto de "variable" es nuevo si vienes del PAWS, pero te será familiar si usas otros lenguajes de

programación. En Inform simplemente se declaran los nombres de las variables, pero no sus tipos, ya que no hay tipos en Inform. En el ejemplo, habría tres variables locales, llamadas `v1`, `v2` y `v3`. Más adelante veremos más cosas sobre variables.
- La lista de variables locales se termina mediante un punto y coma (`;`)
- A continuación viene la lista de instrucciones que componen la rutina. Cada instrucción se separa de la siguiente mediante un punto y coma (`;`) Puedes poner varias instrucciones en la misma línea (separadas por punto y coma), o en líneas diferentes, el verdadero delimitador de instrucciones es el punto y coma, y no el salto de línea, que para Inform es lo mismo que un espacio.
- La definición de la rutina finaliza con el signo `]`
- Al final de la definición, un punto y coma (`;`)

Miremos otra vez el código que has escrito

```
[ Inicializar ;
];
```

En nuestro caso, la rutina se llama `Inicializar`, su lista de variables locales está vacía (pues no hay nada entre el nombre de la rutina y el primer punto y coma), y también está vacío su "cuerpo", es decir, la lista de instrucciones que la componen (puesto que no hay nada entre el primer punto y coma y el signo `]` que da por terminada la rutina).

En esencia: la rutina `Inicializar` que hemos escrito es una rutina *totalmente vacía* que no hace nada de nada. ¿Por qué la ponemos entonces? La razón es que esta rutina tiene que existir en todo programa InformATE, o de lo contrario Inform se negará a crear tu juego. Prueba a quitarla y lo verás.

Para probar a quitar trozos de código mientras estás de pruebas, basta poner un signo de admiración delante de cada línea que quieras quitar. Esto convierte el código en "comentarios", por lo que se hace invisible a ojos de Inform. Si más tarde quieres restituir el código, basta borrar el signo de admiración.

Normalmente esta rutina se ocuparía de especificar cuál es la localidad en que comienza el juego, y de imprimir algún mensaje de bienvenida, pero en este primer esbozo la hemos dejado vacía. Aún así funciona, no es ilegal escribir una rutina que no haga nada, aunque resulta poco útil.

Programando el mapeado

Vamos a escribir el código de cada localidad. En Inform, cada localidad es un objeto. Esto puede sonarte un poco raro si pensabas que los objetos eran cosas como "una piedra", "una espada", etc. Pero es que el concepto de Objeto de Inform es mucho más amplio.

En PAWS se llamaba "objetos" a las cosas que el jugador podía coger y llevarse. En Inform el concepto es muchísimo más amplio:

Para Inform, un objeto es en realidad un *elemento del juego*. Por tanto, los lugares son objetos, los PNJ son objetos, el propio jugador es un objeto, un "tema de conversación" puede ser un objeto. Prácticamente cualquier elemento de los que componen el juego son objetos (excepto las acciones o el vocabulario). Por eso se dice que Inform es *orientado a objetos*.

Aunque todos los objetos tienen siempre una misma estructura, no vamos a entrar aún a describir la estructura general de los objetos, y nos centraremos de momento en la estructura de los objetos que representan lugares. Esta es su estructura:

```
Object nombre_interno "nombre corto"
with   descripcion "Descripción larga de la localidad",
       al_n ... ,
       al_s ... ,
       al_e ... ,
       al_o ... ,
       arriba ... ,
       abajo ... ,
       adentro ... ,
       afuera ... ,
has luz;
```

Es decir:

- La palabra `Object`, que le dice a Inform que aqui va una definición de un objeto. La definición terminará en un punto y coma (que ocurre al final del todo, tras la palabra `luz`)
- Un "nombre interno", que es el que usaremos como programadores para referirnos a cada localidad. Son lo que en PAWS eran el número de localidad. Si te gusta pensar con números, puedes asignar a tus localidades nombres internos como `loc1`, `loc2`, `loc3`, etc... Pero te aconsejo que vayas olvidando la "forma PAWS de pensar" y uses nombres lógicos para referirte a tus localidades, nombres como `PuertaPrincipal`, `Dormitorios`, `EscaleraCaracol`, etc. Estos nombres nunca los verá el jugador. No pueden llevar acentos ni eñes ni espacios. Sólo letras, números y el signo _.
- Un "nombre corto", que es por asi decir el "título" de cada localidad. Esto sí que lo verá el jugador, será de hecho lo primero que se imprima cuando entre en la localidad, y lo que aparecerá en la barra de estado del juego. Puede ser algo como "Puerta Principal", "Dormitorios", "Escalera de Caracol", etc. Debe ir encerrado entre comillas dobles. Por supuesto, aquí sí puedes usar acentos y eñes.
- La palabra especial `with` que indica a Inform que a continuación viene una lista de "propiedades" de esa localidad. Cada propiedad se separa de la siguiente mediante una coma.
- La primera propiedad que ponemos es `descripcion`. Las propiedades son parejas, el primer elemento de la pareja es el nombre de la propiedad (`descripcion`) y el segundo elemento es el valor de esa propiedad (o sea, la descripción en concreto de ese lugar). Entre ambos elementos de la pareja hay un espacio.

  La descripción es normalmente un texto, por lo que debemos encerrarlo entre comillas dobles. El texto puede ser muy largo por lo que ocuparía varias líneas de tu programa, no obstante los lugares por donde cortes las líneas no influyen en lo que el jugador verá. Su intérprete cortará las líneas siempre por el lugar adecuado según el ancho de su pantalla. Si quieres forzar un salto de línea, debes usar el signo ^, como en la constante `Titular`).

  Recuerda que una vez terminado el texto y cerradas las comillas, debes poner una coma para separar esta propiedad de las siguientes.

- La siguiente propiedad se llama `al_n` y su valor indica a dónde se llega desde este lugar si se sale por el norte. Los puntos suspensivos que hay después están representando el nombre interno de otra localidad. Y detrás una coma.
- De forma análoga, las propiedades `al_s`, `al_e`, etc indican a dónde llevan las restantes salidas de la localidad. No es necesario listar todos los puntos cardinales, sino sólo los que realmente llevan a

algún sitio. Los que no se listan, se entiende que son direcciones sin salida. (También existen las direcciones `al_no`, `al_so`, `al_ne` y `al_se` no usadas en el ejemplo)

- Una vez finalizada la lista de propiedades, se inicia la lista de atributos, la cual comienza por la palabra especial `has` (que en inglés significa "tiene").
- Los atributos son como ciertos flags de PAWS, se trata de indicadores para señalar cosas como si el objeto es comestible, si es una prenda, si da luz, etc... En el caso de las habitaciones, el atributo más usual es el llamado `luz`, que como puedes suponer indica que el lugar está iluminado. ¡Si olvidaras poner ese atributo el jugador no podría ver la descrcipción de la localidad a menos que llevara consigo algún objeto con `luz`!

Una vez comprendida la estructura de las localidades, codificar el mapa es pan comido. Copia el siguiente código en la zona donde estaba el comentario "Aquí irían las definiciones de las localidades y los objetos del juego". Observa que, de momento, ponemos conexión "normal" en lugar de la puerta secreta.

```
Object  PuertaPrincipal "Puerta Principal"
 with   descripcion "Estás junto a la puerta principal.
            A su lado puedes ver una mesa de guardia y en la pared
            norte hay una chimenea.",
        al_e Dormitorio,
 has    luz;

Object  Dormitorio "Dormitorio"
 with   descripcion "Varios maltrechos catres se amontonan en esta
            habitación.",
        al_o PuertaPrincipal,
        al_e EscaleraCaracol1,
 has    luz;

Object  EscaleraCaracol1 "Escalera de caracol"
 with   descripcion "El viento ulula a través de la empinada
            escalera de caracol, una vieja armadura parece vigilar
            la escalera.",
        arriba EscaleraCaracol2,
        al_o Dormitorio,
 has    luz;

Object  Mazmorra "Mazmorra"
 with   descripcion "Una silenciosa estancia débilmente alumbrada
            por los rayos de luna que se filtran a través de un
            pequeño ventanuco. El suelo está lleno de paja, colgando
            de unos grilletes en la pared observas un esqueleto
            humano.",

        al_e EscaleraCaracol2,
 has    luz;

Object  EscaleraCaracol2 "Escalera de caracol"
 with   descripcion "Los desgastados peldaños de piedra resbalan en
            ocasiones. A mitad de la escalera una antorcha en la
            pared impide que la oscuridad sea completa.",
        abajo EscaleraCaracol1,
        al_o Mazmorra,
        arriba AltoTorre,
 has    luz;

Object AltoTorre "Alto de la torre"
 with   descripcion "Una gran cama preside la estancia, los gruesos
            barrotes no permiten la salida por la ventana, aunque de
            todos modos estaría demasiado alta.",
```

```
       abajo EscaleraCaracol2,
has    luz;
```

Unas notas sobre el código anterior:

- No puede haber en el juego dos objetos con el mismo nombre interno. Por eso he llamado a una parte de la escalera `EscaleraCaracol1` y a la otra `EscaleraCaracol2`.
- Sin embargo nada impide que tengan el mismo "nombre corto", simplemente el jugador verá el mismo "título" en ambas localidades.

Ahora será necesario también modificar la rutina `Inicializar` para indicar en cuál de esas localidades comienza el juego.

En Inform, existe una *variable global* llamada `localizacion` que contiene siempre la localidad en la que se halla el jugador. El concepto de variable global es muy similar al concepto de "Bandera" de PAWS, si bien en PAWS las banderas se identificaban mediante un número (así la bandera que indicaba dónde está el jugador, era la bandera número 254), mientras que en Inform se identifican mediante un nombre (`localizacion` es el nombre de esta "bandera"). Otras importantes diferencias son que en PAWS el número de banderas estaba limitado a 255, y no era necesario "declararlas" de antemano, mientras que en Inform no hay límite al número de variables globales, pero es necesario declararlas de antemano (es decir, "anunciar" a Inform de que vamos a crear una nueva variable global, indicándole el nombre que vamos a darle). La variable `localizacion` es declarada en `EParser.h`, el fichero que hemos incluido al principio. Otra diferencia importante es que en PAWS una bandera sólo podía servir para almacenar un número entre 0 y 255, mientras que en Inform puede usarse para almacenar un número entre −32768 y +32767, o también una palabra, una frase, un objeto, un lugar.. ¡cualquier cosa!

Para hacer que el lugar de comienzo sea la puerta principal, basta cargar en la variable `localizacion` el objeto `PuertaPrincipal`. La carga de una variable es algo análogo al condacto `LET` del PAWS, pero en Inform se usa la siguiente sintaxis:

```
  variable = valor;
```

Así, por ejemplo:

```
  v2 = 32;
```

Guardaría el número 32 dentro de la variable `v2`, de forma análoga a lo que hacía el condacto `LET`, mientras que:

```
  v1=v2;
```

Copiaría el valor de la variable `v2` en la variable `v1`, de forma análoga a lo que hacía el condacto `COPYFF`

De modo que ahora debes modificar la rutina `Inicializar` para que ponga:

```
[ Inicializar ;
   localizacion = PuertaPrincipal;
];
```

Observa que ahora `Inicializar` ya no está vacía. Tiene una sola instrucción que sirve para asignar un valor a la variable `localizacion`. Recuerda que la instrucción debe terminar con punto y coma.

Otra interesante variable de Inform es una llamada `modomirar`, que controla si la descripción de la localidad debe repetirse o no cada vez que el jugador vuelve a entrar en una localidad en la que ya había estado. El valor por defecto de esta variable implica que la descripción sólo se le muestre una vez, y si desea volver a verla debe escribir el verbo MIRAR. Si prefieres que la descripción se muestre siempre, debes hacer que esta variable tenga el valor 2 cuando el juego comience, es decir, modifica `Inicializar` para que sea:

```
[ Inicializar ;
   localizacion = PuertaPrincipal;
   modomirar=2;
];
```

Es importante que te acostumbres desde el primer momento a "comentar" tu código, para que otros puedan comprenderlo y para que tú mismo lo entiendas cuando lo mires unas semanas más tarde. Añadir comentarios al juego no lo hace crecer de tamaño ¡recuerda que Inform ignora tus comentarios! Así que una rutina `Inicializar` más "profesional" sería así:

```
[ Inicializar ;
    ! Lugar de comienzo: la puerta principal
    localizacion=PuertaPrincipal;

    ! Activamos esta variable para que la descripción de los lugares
    ! se repita siempre, incluso si el jugador ya ha estado allí
    modomirar=2;
];
```

Es habitual también incluir en la rutina `Inicializar` algún mensaje de bienvenida al juego, o una introducción para situar al jugador. El comando para mostrar texto en Inform es `print`, seguido del texto en cuestión encerrado entre comillas dobles. Por ejemplo:

```
  print "¡¡Bienvenido a mi primer juego!!^^";
```

Como ya sabes, el signo ^ representa un salto de línea. Poniendo dos seguidos dejamos una línea en blanco debajo del texto anterior. Observa que en Inform no hay tablas de mensajes de usuario. No necesitas guardar tus textos en una tabla para después acceder a ellos mediante un número, sino que el texto lo pones directamente a continuación del print. No olvides encerrarlo entre comillas.

Los juegos conversacionales escriben mucho texto. En juegos complicados esto significa que habrá que usar muchas veces la orden `print`, por ello Inform permite una abreviatura especialmente útil, pero un poco peligrosa para los principiantes. Consiste en no poner `print`, sino simplemente el texto entre comillas. Cuando Inform encuentra un texto entrecomillado en un lugar donde esperaba una instrucción (o sea, dentro de una rutina), lo que hará será imprimir ese texto como si hubieras puesto `print` delante, añadir un salto de línea al final (como si hubieras puesto un ^ justo antes de cerrar las comillas), *y terminar la ejecución de la rutina* (algo equivalente al DONE del PAWS).

Esto es muy importante, merece la pena repetirlo: una cadena entrecomillada en lugar de una instrucción equivale a un `print` de la cadena, seguido de un salto de línea y seguido del retorno inmediato de la rutina en que aparece.

Por tanto, si queremos escribir un mensaje de bienvenida desde nuestra rutina `Inicializar`, tenemos varias posibilidades. En primer lugar, usando `print` (forma recomendada a los principiantes):

```
[ Inicializar;
    ! Lugar de comienzo: la puerta principal
    localizacion=PuertaPrincipal;

    ! Activamos esta variable para que la descripción de los lugares
    ! se repita siempre, incluso si el jugador ya ha estado allí
    modomirar=2;

    ! Mensaje de bienvenida
    print "¡¡Bienvenido a mi primer juego!!^^";
];
```

La orden `print` puede aparecer en cualquier lugar dentro de `Inicializar`, a fin de cuentas da lo mismo imprimir la bienvenida antes o después de haber inicializado las variables. La segunda forma, para programadores más expertos, es usar la abreviatura en la que no se pone `print`. Quedaría así:

```
[ Inicializar;
    ! Lugar de comienzo: la puerta principal
    localizacion=PuertaPrincipal;

    ! Activamos esta variable para que la descripción de los lugares
    ! se repita siempre, incluso si el jugador ya ha estado allí
    modomirar=2;

    ! Mensaje de bienvenida
    "¡¡Bienvenido a mi primer juego!!^";
];
```

En este caso, observa con atención los detalles siguientes:

- La cadena tiene ahora solo un ^ en su interior, en vez de dos, ya que cuando se omite `print`, siempre se añade un salto de línea automáticamente al final del texto.
- La instrucción de imprimir *debe ser la última de la rutina*, puesto que justo después de haber impreso eso, la rutina se abandona, sin ejecutar lo que hubiera debajo. Si hubiésemos colocado el texto de bienvenida al principio de la rutina, las instrucciones que inicializan `localizacion` y `modomirar` no llegarían a ser ejecutadas nunca (Inform te avisa de esto al compilar).

Puedes bajarte el código fuente con todo esto ya tecleado aqui. También se incluye una versión ya compilada para que puedas jugarla y comprobar cómo puedes deambular por el mapa.

Prueba a cambiar la descripción de una localidad, no tengas miedo de experimentar, es la única forma de aprender. ¿Cómo modificarías el programa para que bajando desde la `PuertaPrincipal` se llegue a un `Sotano`? ¡Hazlo! ¿Cómo eliminarías la conexión que hay entre la `EscaleraCaracol1` y la `Mazmorra`? ¡Hazlo!

Entendiendo Inform (y olvidando PAWS)

Tenemos bastantes rompecabezas en nuestro mini−juego, y esto va a involucrar muchas cosas nuevas: definir objetos, definir los comportamientos de esos objetos, etc... Para comprender realmente cómo se hacen estas cosas en InformATE es necesario conocer antes su filosofía, que es diferente en muchas cosas a la de PAWS. Por ello iremos muy despacio en el primer rompecabezas, y ya correremos más en los siguientes.

El primer rompecabezas que vamos a implementar será la puerta secreta. De momento, vamos a quitar la conexión que había entre la escalera y la mazmorra. Vamos, edita tu fichero y borra la línea que dice `al_o`

`Mazmorra` dentro del objeto `EscaleraCaracol2`. ¿Ya lo has hecho? Compila el juego y juegalo de nuevo. Verás como ya no se puede ir al oeste desde la escalera. Sin embargo, sí que se podría aún ir al este desde la Mazmorra (¡si hubiera alguna forma de entrar en la mazmorra!), puesto que no hemos borrado esa salida. Inform, al igual que PAWS, permite conexiones asimétricas o ilógicas.

Vamos a hacer un poco de trampa. Durante la fase de programación y pruebas puede resultar útil tener ciertos "superpoderes", que te permitan teleportarte y hacer otras maravillas dentro del juego. Muchos de estos "superpoderes" están activados por defecto en nuestro juego. Arranca el juego y escribe "VERSION", aparecerá el título del juego, y después información sobre la versión del juego y de la librería. ¿Aparecen las letras SD al final de esta línea? Si es así, tenemos los superpoderes (en realidad, esto indica que está activado el "modo Debug"). En la versión final del juego esto deberá estar desactivado, más adelante diremos cómo desactivarlo.

Uno de los superpoderes es el de teletransportación. Para usarlo necesitas conocer el número de la localidad a la que vas a teletransportarte (¡como en el MUD!). Este es un número que Inform asigna a cada objeto del juego y que normalmente el programador no conoce. Podemos averiguarlo con el comando XLISTA, el cual nos mostrará todos los objetos del juego, y al lado de cada localidad, entre paréntesis, su número secreto. [NOTA: XLISTA sólo está disponible a partir de la versión 001101 de InformATE]. En nuestro juego saldrá algo como esto:

```
> XLISTA
Oscuridad (20)
Puerta Principal (25)
  a ti mismo
Dormitorio (26)
Escalera de caracol (27)
Mazmorra (28)
Escalera de caracol (29)
Alto de la torre (30)
```

Aqui puedes ver que, además de las localidades del juego, hay dos objetos misteriosos. Uno llamado "Oscuridad" (con el número 20) y otro llamado "a ti mismo" (sin número). La Oscuridad es un lugar ficticio que existe en todo juego InformATE, al que vas a parar tan pronto como te quedes a oscuras.

"A ti mismo" es el jugador. ¡Ya te había dicho que el jugador es un objeto más! Además, fíjate que esa línea aparece un poco metida hacia la derecha, justo debajo de la línea que dice "Puerta Principal", lo que significa que en este momento, el objeto jugador está dentro del objeto `PuertaPrincipal`.

Además hemos obtenido información sobre los números de las diferentes localidades, con lo que ya podemos teleportarnos a cualquiera de ellas. ¡Incluso a la Mazmorra a pesar de que no tenía entrada posible! Para ello basta poner el comando `XIR 28`. Si vuelves a usar `XLISTA` a continuación, verás como "a ti mismo" está ahora dentro de la `Mazmorra`.

Comprueba, una vez teleportado, que puedes salir de la mazmorra si te mueves al este (irás a parar a la escalera de caracol), pero no puedes volver a entrar en la mazmorra yendo hacia el oeste, pues habiamos borrado esa conexión.

Lo que queremos ahora es que la conexión de entrada a la Mazmorra reaparezca mágicamente cuando el jugador intente EMPUJAR la antorcha, y desaparezca de nuevo cuando intente TIRAR DE la antorcha.

Para ello tenemos que crear el objeto antorcha. Esto es otra importantísima diferencia entre Inform y PAWS. En PAWS el jugador podía referirse a cosas que en realidad no existían como objetos dentro del juego (eran meros NOMBRES, pero no objetos). En Inform esto no es posible. Cualquier cosa a la que el jugador deba referirse, ha de ser un objeto.

Para comprender por qué, es necesario entrar a explicar con un poco de detalle las diferencias entre el parser de PAWS y el parser de Inform.

La labor del parser: Acciones y Objetos

El parser es la parte del juego que lee una frase que el jugador ha escrito y la descompone en partes, tratando de asignar un significado a cada parte.

El parser de PAWS era muy sencillo, ya que lo único que hacía era comparar cada palabra de las que el jugador ha escrito con las palabras que el juego había definido en su vocabulario, tras lo cual asignaba una *categoría* diferente a cada una de estas palabras, ignorando sin más las no reconocidas. De este modo, al final del parsing se tenían en ciertas banderas:

- Qué verbo ha usado el jugador
- Qué nombre ha usado como primer nombre (y qué adjetivo)
- Qué nombre ha usado como segundo nombre (y qué adjetivo)
- Qué preposición ha usado.

A partir de aquí, el resto del trabajo correspondía al juego, que debía comparar el verbo usado por el jugador con cada posible verbo del juego, una vez encontrado, debía comparar el nombre usado por el jugador con cada posible nombre del juego, y una vez localizado, debía averiguar si ese nombre tenía sentido en la situación actual del jugador. Así, por ejemplo, ante la frase TIRA DE LA ANTORCHA, el parser simplemente se limita a decirnos que Verbo=TIRA, preposicion=DE, Nombre1=ANTORCHA. El programador debía hacer una serie de comprobaciones de este estilo:

```
Si Verbo es TIRA
  y Nombre1 es ANTORCHA
     Entonces:
         Si el jugador esta en la escalera de caracol
             ....
             .... Instrucciones para abrir la puerta secreta
             ....
         Si no
             mostrar el mensaje ``No veo ninguna antorcha aqui''
```

En Inform, la mayor parte de estas comprobaciones las hace la librería. El programador debe escribir tan solo la parte que correspondería a las instrucciones para abrir la puerta secreta.

Pero antes de explicar cómo hace la librería para interpretar la frase, remarquemos dos conceptos clave de Inform: ACCIONES y OBJETOS. Estos conceptos marcan toda la diferencia entre Inform y los parsers tipo PAWS, mientras no comprendas estos conceptos, no entenderás Inform. Repíte estas dos palabras todas las noches hasta quedar dormido, ACCIONES y OBJETOS, hasta que logres olvidarte de "verbos y nombres".

Acciones

El jugador escribe VERBOS en sus FRASES. Sin embargo, lo que el juego recibe son ACCIONES. El Parser se ocupa de transformar FRASES en ACCIONES.

Un mismo verbo puede dar lugar a diferentes acciones, según la frase en que aparece. Fíjate: TIRA LA PIEDRA, TIRA DE LA ANTORCHA, TIRA LA PIEDRA CONTRA EL ESPEJO. El mismo verbo "TIRA", tiene significados bien distintos. En el primer caso significa algo como "dejar caer", mientras que en el segundo es más bien "dar un tirón" y en el tercero es "lanzar" o "arrojar".

El concepto de *accion*, pues, no es más que el significado de un verbo. El cometido del parser de Inform es analizar frases y convertirlas en acciones, es decir, *extraer el significado* de la frase. En la primera frase, "TIRA LA PIEDRA", la acción que genera el parser es `Dejar`, en la segunda frase "TIRA DE LA ANTORCHA" la acción que genera el parser es `Tirar`, y en la tercera frase "TIRA LA PIEDRA CONTRA EL ESPEJO", la acción generada es `Lanzar`.

Es muy importante no confundir los verbos con las acciones. En los tres ejemplos anteriores el verbo ha sido el mismo "TIRAR", pero las acciones generadas han sido diferentes. Y la inversa también es cierta, hay muchos verbos diferentes que darán lugar a la misma acción. Así, la acción `Dejar` puede ser generada por frases bien distintas: "DEJA LA PIEDRA", "SUELTA LA PIEDRA", "TIRA LA PIEDRA". Y la acción `Lanzar` también puede ser generada por diferentes frases: "LANZA LA PIEDRA", "ARROJA LA PIEDRA", "TIRA LA PIEDRA CONTRA LA PARED".

En PAWS el parser no intentaba "comprender" la frase, en Inform sí, y una vez comprendida, da lugar a una ACCION.

En PAWS existían verbos sinónimos, en Inform lo que ocurre es que muchas frases distintas pueden dar lugar a la misma ACCION. Podría decirse que en Inform hay "frases sinónimas".

¿En qué se basa el parser para decidir que "TIRA DE LA ANTORCHA" tiene distinto significado que "TIRA LA PIEDRA"? No se conforma con examinar el verbo, sino que tiene que leer la frase completa para tratar de extraer un significado. En este ejemplo, el parser ha sido instruido para que, ante una frase que use TIRA seguida de la palabra DE, seguida de un nombre de objeto, la acción que genere sea `Tirar`. Si, en cambio, la frase es simplemente TIRA seguida de un nombre de objeto (sin el DE), la acción será `Dejar`. Finalmente, si la frase tiene la estructura TIRA <nombre de objeto> CONTRA <nombre de objeto>, entonces la acción será `Lanzar`.

Esta información de qué acción debe generar para cada posible estructura de frase, no está incrustada dentro del parser de forma que sea imposible cambiarla. ¡Todo lo contrario! Se halla definida en un fichero aparte (`Gramatica.h`) y es bastante sencillo cambiarla, de modo que si tu lo prefieres, puedes hacer que TIRA PIEDRA genere la acción `Lanzar` en lugar de `Dejar`.

Por supuesto también es posible crear nuevas acciones, y asignarlas a nuevas estructuras de frase. Así, puedes crear la acción `Ayuda` (que no existe por defecto en la librería), y hacer que esta acción se genere ante las frases "AYUDA", "SOCORRO", "HELP". No obstante, la definición acciones y frases nuevas se dejará para más adelante.

La librería InformATE ya trae predefinidas un montón de frases y las correspondientes acciones para cada caso. Es importante que conozcas qué acciones se generan para cada verbo, ya que como programador tú te las verás únicamente con las acciones, y no con los verbos.

Más adelante daremos un truco para saber qué acciones se generan ante cada posible frase, de modo que no necesites saber eso de memoria, ni andar consultando manuales.

Objetos

La otra palabra clave que debes comprender es la del objeto, ya que:

Las acciones actúan sobre los objetos.

En PAWS, se programaban "tablas de respuestas" que indicaban qué responder para cada verbo, ante cada posible nombre. En inform en cambio los programas van "dentro" de los propios objetos, e indican qué debe responder ese objeto ante cada posible acción que se intente sobre él. Las acciones que el objeto no contemple, serán manejadas de forma automática por la librería.

Basta de teoría. Programemos de una vez el objeto antorcha para ver qué ocurre con él. Añade el siguiente código en tu juego. Puedes escribirlo en cualquier punto después del objeto `EscaleraCaracol2`, incluso podría ir al final del fichero. Sin embargo es buena idea ponerlo justo detrás de la `EscaleraCaracol2`, ya que de ese modo los lugares y los objetos que hay en ellos quedan juntos en el código.

```
Object antorcha "antorcha" EscaleraCaracol2
 with   nombre 'antorcha',
 has    femenino;
```

Fijate en la estructura de un objeto, en realidad es casi idéntica a la estructura de una localidad:

- La palabra especial `Object` indica que va a comenzar la definición de un objeto. Esta definición terminará cuando aparezca un punto y coma (el cual aparece tras la palabra `femenino`).
- El nombre interno del objeto. Esto es equivalente al número de objeto de PAWS. Si te gusta pensar con números, puedes darle como nombre interno algo como `Objeto1`, `Objeto2`, etc.. Pero admitirás que resulta mucho más lógico llamarlo `antorcha`. Este nombre nunca lo verá el jugador (y no puede contener acentos ni eñes, ni coincidir con el nombre interno de ningún otro objeto del juego)
- El nombre corto del objeto, que debe escribirse entre comillas dobles. Este es el nombre que el juego mostrará cuando mencione el objeto (como en "Llevas una antorcha", o "Puedes ver una antorcha", etc)
- Y ahora, como novedad, aparece el nombre de otro objeto. En este caso se trata del objeto `EscaleraCaracol2`. Este otro objeto será el "recipiente" que contiene a la antorcha. Si es una localidad, esto significará que la antorcha está en esa localidad.
- La palabra `with` que da inicio a la lista de propiedades, al igual que en las localidades.
- La propiedad más importante que debe tener todo objeto, es su nombre. Te preguntarás ¿pero el nombre no era "antorcha", que ya habíamos puesto en la línea anterior? Y la respuesta es: no, "antorcha" es el nombre que usa la librería cuando tiene que emitir mensajes relacionados con este objeto. Sin embargo `nombre` es la propiedad que indica qué nombres puede usar el jugador para referirse a este objeto.

En este caso particular, el nombre por el que el jugador se referirá será seguramente 'antorcha', pero este es el lugar donde podríamos poner más sinónimos (como por ejemplo 'tea'). Si quieres poner sinónimos, irían separados por espacios:

```
with nombre 'antorcha' 'tea',
```

Los nombres que pongas en la propiedad `nombre` deben ir entre comillas simples (apóstrofes). Todas estas palabras las va recopilando Inform a medida que compila tu programa y las va metiendo en el Diccionario del juego.

Inform no tiene una sección donde se le indique el vocabulario del juego, sino que él mismo lo va construyendo a medida que va encontrando palabras entre comillas simples como las anteriores.

En estas palabras puedes poner acentos o eñes, pero **se recomienda que no las pongas**. Piensa que los juegos Inform pueden ser ejecutados en muchos ordenadores diferentes, algunos de los cuales ni siquiera tienen eñe en su teclado. Si pones en el `nombre` por ejemplo la palabra 'niño', estás forzando al jugador a que use esa palabra para referirse al objeto, ¡y tal vez el jugador no pueda escribirla! Si en cambio, usas 'nino' (con ene en lugar de eñe), todo el mundo puede escribirlo. Además, el juego comprenderá también la palabra 'niño' (con eñe), porque el parser convierte todas las eñes en enes si la palabra que puso el jugador no está en su diccionario. Lo mismo es aplicable para las vocales acentuadas.

- La lista de propiedades es corta en este caso, pues sólo le hemos puesto la propiedad `nombre`. Más adelante le daremos más.
- Finalmente, viene la lista de atributos que comienza por la palabra especial `has`.
- Como mínimo, un objeto ha de tener atributos que indiquen su género y número, para que InformATE sepa qué artículo debe usar delante del objeto cuando se refiera a él. La antorcha es `femenino`. Algunos objetos pueden llevar también el atributo `nombreplural` (por ejemplo "gafas").

  Si el género es masculino, puedes optar por poner el atributo `masculino` o no ponerlo (InformATE presupone atributo `masculino` si no se pone).

- El punto y coma final da por terminada la definición del objeto.

Puedes bajarte aqui el código fuente del programa que incluye la antorcha, y su versión lista para jugar. Vamos a hacer muchos experimentos con esta antorcha, de modo que te aconsejo que sigas las explicaciones con el juego funcionando. Si pones VERSION en este juego precompilado, observarás que al final ya no aparecen las letras `SD`, sino sólamente `D`. La letra `D` indica que aún estamos en "modo debug" y por tanto seguimos disponiendo de *superpoderes*. La letra `S` indicaba además otro modo especial que no interesa para estas pruebas, y que es mejor desactivar. Para lograr desactivar el modo `S`, dejando activado el modo `D`, debes usar la siguiente orden al compilar:

```
INFORMBP -D -~S torre3.inf
```

Vamos con la primera prueba. Teclea XLISTA para ver cómo el nuevo objeto realmente forma parte del juego:

```
>xlista
Oscuridad (20)
```

```
Puerta Principal (25)
  a ti mismo
Dormitorio (26)
Escalera de caracol (27)
Mazmorra (28)
Escalera de caracol (29)
  una antorcha
Alto de la torre (31)
```

¿Has visto? la antorcha está en la localidad número 29, la escalera de caracol. Vamos a teleportarnos allí. Puedes usar como antes el verbo `XIR 29`, pero ahora que la localidad contiene un objeto, puedes usar otra forma más cómoda: `IRDONDE ANTORCHA`, lo cual te telportará a la habitación en la que esté el objeto que tiene 'antorcha' en su `nombre`. [Nota, debido a un bug, este comando sólo funciona correctamente en InformATE! 6/10 a partir de la revisión 001101]

```
> IRDONDE ANTORCHA
Escalera de caracol
Los desgastados peldaños de piedra resbalan en ocasiones. A mitad de la escalera
una antorcha en la pared impide que la oscuridad sea completa.

Puedes ver una antorcha.
```

Efectivamente, el mensaje "puedes ver una antorcha" nos ratifica que está en el lugar correcto. Realmente ese mensaje sobraba, puesto que la descripción de la localidad ya había mencionado la antorcha. Esto es muy fácil de corregir y después veremos cómo. Por el momento vamos a realizar algunas acciones sobre la antorcha.

¡Y este es un truco imprescindible! Otro de los superpoderes de que disponemos en modo "Debug" es el poder ver las *acciones* que genera el parser ante cada posible frase que tecleamos. Para ello pon:

```
> ACCIONES
[Listado de acciones activado.]
```

Y ahora intenta diferentes verbos sobre la antorcha, para averiguar qué acción genera cada uno. Como hemos dicho, el programador necesita conocer siempre las acciones generadas, pues la programación del juego ocurre en torno a las acciones (y no a los verbos como en PAWS).

Comprobemos que todo lo que os había contado antes sobre el verbo "TIRAR" es cierto, intentando las diferentes formas de este verbo y comprobando cómo cada una da lugar a una acción diferente:

```
>tirar antorcha
[ Acción Dejar con uno 30 (antorcha) ]
Para dejar la antorcha deberías tenerla.
```

El mensaje entre corchetes ha aparecido porque hemos puesto antes `ACCIONES`. El mensaje que aparece a continuación es la respuesta estándar cuando intentamos dejar un objeto que no tenemos. ¿Qué significa el mensaje "[Acción `Dejar` con `uno` 30 (antorcha)]"?

En Inform casi todas las acciones recaen sobre algún objeto, aunque hay algunas excepciones (por ejemplo `Saltar`, `Cantar`, `Rezar` son acciones que no necesitan objeto, mientras que `Coger`, `Dejar` etc son acciones que siempre han de ir referidas a un objeto: el objeto que se coge o se deja). Incluso en algunos casos se involucran dos objetos (como en la acción `Meter` que requiere el objeto que se está metiendo y el objeto dentro del cual se está metiendo).

El parser, después de analizar la frase y tras haber determinado el objeto (u objetos) sobre los que la acción tiene lugar, deja preparadas tres variables:

- `accion`: esta variable contiene la acción que se ha generado
- `uno`: esta variable contiene el primer objeto relacionado con esa acción
- `otro`: esta variable contiene el segundo objeto relacionado con esa acción

En cierta manera, es algo análogo al verbo, el nombre1 y el nombre2 de PAWS, solo que en Inform no se trata de palabras, sino de acciones y objetos. El parser siempre intentará relacionar las palabras con los objetos del juego, y si no puede, causará un mensaje de error.

Volvamos a lo que ha dicho el juego: "Acción `Dejar` con `uno` 30 (antorcha)" Su significado debe de estar más claro ahora. La acción generada es `Dejar` (aunque el verbo usado ha sido "TIRA"), y el objeto sobre el que recae la acción (lo que Inform llama "uno") es el objeto número 30 (que es la antorcha como nos aclara después entre paréntesis). El número de objeto no es importante para nosotros, es un número interno que Inform asocia a cada objeto. [Jeje, date cuenta que ahora que sabemos el número de la antorcha podríamos teleportarnos a su interior con `XIR 30` ¿qué crees que pasará?]

Otro detalle importantísimo. Si el parser no es capaz de asociar la frase del jugador con un objeto válido en las cercanías, el parser se detiene con un error y ***la acción no llega a generarse***.

Esto se comprueba fácilmente en el juego. Ve al piso inferior (donde la antorcha ya no es visible) e intenta de nuevo poner "Tira antorcha". Veamos qué ocurre:

```
>tira antorcha
No veo eso que dices.
```

No aparece ningún mensaje entre corchetes indicando la acción ¡porque no ha llegado a generarse ninguna acción! El comando ha sido abortado porque no se ha encontrado el objeto sobre el cual se supone que debería actuarse.

Esto es otra diferencia fundamental con PAWS, en donde el parser no intentaba asociar los nombres con los objetos, sino que se limitaba a dejarlos en una variable. Esto permitía que el juego pudiera reaccionar a nombres de cosas, sin que esas cosas existieran en realidad. En Inform esto no es posible. Cualquier cosa a la que el jugador deba referirse, debe ser un objeto del juego.

Sigamos experimentando con las acciones, viendo qué entiende el parser ante diferentes comandos:

```
>salta
[ Acción Saltar ]
Saltas en el sitio, sin ningún resultado.
```

Fíjate cómo en este caso la acción que se genera no lleva objeto asociado (no se menciona el valor de `uno`). ¿Y si intentamos del mismo modo poner "DEJA" sin más palabras?

```
>deja
¿Qué quieres dejar?
```

No se ha generado la acción `Dejar`, porque el parser detecta que este es un comando incompleto. El parser ha sido instruído para que sepa que detrás del verbo DEJA debe ir siempre el nombre de un objeto. Como le falta esto, la acción ni siquiera es iniciada.

Volvamos al piso de arriba e intentemos algunas acciones más sobre la antorcha.

```
>coge antorcha
```

```
[ Acción Coger con uno 30 (antorcha) ]
Cogida.
```

Vaya. ¿Podemos coger la antorcha? Esto no debería ser asi, el jugador no debería poder llevarsela. Lo que ha ocurrido es que, dado que aún no hemos programado nada en el objeto `antorcha`, la librería ejecuta las acciones por defecto. Y por defecto, la librería permite coger cualquier objeto que se halle al alcance del jugador. Veremos después cómo impedirlo.

Podríamos seguir experimentando, pero te dejaré que lo hagas tú mismo. Intenta diferentes verbos sobre la antorcha. Prueba cosas absurdas, intenta abrirla, cerrarla, romperla, quemarla, olerla, lanzarla al suelo, empujarla, tirar de ella, girarla, moverla. Verás que Inform comprende todas esas frases y las convierte en diferentes acciones (o a veces en la misma, en particular fíjate en la acción que se genera para MOVER ANTORCHA).

Ahora pasemos a algo más interesante. Vamos a programar la antorcha para que "reaccione" ante ciertas acciones, evitando así que se ejecute la acción por defecto para esos casos.

Cómo se ejecutan las acciones

Ya que queremos que en nuestro juego pasen cosas cuando el jugador empuje la antorcha, lo primero que debemos averiguar es la acción que se generará ante la frase "EMPUJA ANTORCHA". Para ello podemos usar la depuración de acciones como hemos hecho antes. Un buen juego deberia contemplar además otras posibles ideas que se le pueden ocurrir al jugador, como "EXAMINA ANTORCHA", "TIRA DE LA ANTORCHA", "MUEVE LA ANTORCHA", "GIRA LA ANTORCHA", etc. Así que aprovechemos ahora para indagar qué acciones se generan para cada una de esas frases:

```
>empuja antorcha
[ Acción Empujar con uno 30 (antorcha) ]
No ocurre nada, aparentemente.

>examina antorcha
[ Acción Examinar con uno 30 (antorcha) ]
No observas nada especial en la antorcha.

>tira de la antorcha
[ Acción Tirar con uno 30 (antorcha) ]
No ocurre nada, aparentemente.

>mueve la antorcha
[ Acción Empujar con uno 30 (antorcha) ]
No ocurre nada, aparentemente.

>gira la antorcha
[ Acción Girar con uno 30 (antorcha) ]
No ocurre nada, aparentemente.
```

Bueno, hemos averiguado que hay al menos tres acciones que deberíamos tener en cuenta en nuestro juego: `Empujar`, `Tirar` y `Girar` (fíjate que el verbo MOVER produce también la acción `Empujar`, esto es porque en el fichero `Gramatica.h` se ha definido así, aunque puede cambiarse. La asignación de verbos a acciones es programable).

Por otra parte `Examinar` debería dar un mensaje un poco más descriptivo. Todo esto implica *programar la antorcha*.

En PAWS las respuestas a los verbos se codificaban en el proceso llamado "tabla de respuestas". En Inform, en cambio, se codifican en cada objeto.

Cada objeto lleva "incrustadas" las respuestas ante las posibles acciones.

Cuando el parser recibe una frase como "EMPUJA ANTORCHA", da los siguientes pasos:

- Analiza la frase, buscando en ella una estructura y comparándola con todas las estructuras que tiene pre−programadas. En particular, encontrará que la frase encaja con una escructura tipo "EMPUJA <algo>", donde <algo> ha de ser el nombre de un objeto que esté en las cercanías.
- Realizará entonces un bucle en el que recorrerá todos los objetos de las cercanías, a ver si encuentra alguno que pudiera ser una "antorcha". Esta indagación suele ser tan simple como comparar el `nombre` de cada objeto con la palabra "antorcha".
- Si no encuentra ningún objeto plausible, el parser inmediatamente finaliza, con un mensaje de error como "No veo eso que dices", y la acción no llega a generarse (el mensaje en concreto es programable también, pero este es el mensaje por defecto).
- Si encuentra un objeto llamado "antorcha", entonces el parsing ha tenido éxito. La variable `accion` toma el valor `Empujar`, y la variable `uno` toma el valor 30 (que representa al objeto `antorcha` en nuestro juego).
- Comienza la *secuencia de actuación*. En la secuencia de actuación, la librería da los siguientes pasos:
    - "Avisa" a todos los objetos que hay cerca, de que se va a intentar la acción `Empujar` con `uno=antorcha`. Cualquier objeto puede abortar la acción. Enseguida explicaremos cómo tiene lugar este "aviso" y cómo un objeto puede abortar la acción. Esto es útil, por ejemplo, para lograr que un PSI cercano impida ciertas acciones (recuerda que un PSI es un objeto para Inform)
    - Si los objetos cercanos no ponen inconveniente alguno, la acción progresa un poco más, y entonces la librería "avisa" al objeto `antorcha` de que el jugador acaba de realizar la acción `Empujar` sobre ella. La `antorcha` debería tener preparada una respuesta para este caso.
    - Si la `antorcha` tiene preparada respuesta, se ocupará ella misma de continuar la acción. Aquí tiene lugar la programación de lo que ocurre al empujar la antorcha (la aparición de la salida oeste). La librería ya no hará nada más con respecto a esta acción. Es algo así como si la antorcha hiciera un DONE.
    - Si la antorcha no tiene preparada respuesta, entonces la librería ejecutará su acción `Empujar` por defecto. La acción `Empujar` por defecto se limita a escribir "No ocurre nada, aparentemente" (el mensaje también puede ser redefinido)
    - La acción `Empujar`, en particular, no hace nada más, puesto que no se ha cambiado el estado del juego. Todos los objetos permanecen exactamente igual a como estaban antes. Sin embargo otras acciones sí que cambian el juego. Por ejemplo `Coger` o `Dejar` cambia de sitio algunos objetos. En este tipo de acciones, la librería "avisa" de nuevo al objeto en cuestión de que la acción `Coger` o `Dejar` ha finalizado con éxito, por si el objeto quiere hacer algo, antes de que la librería imprima finalmente un mensaje indicador del éxito de la accio (como "Cogido", o "Dejado").

Esta forma de funcionar, aunque puede parecer complicada al principio, te libera en realidad de un montón de trabajo. Tu programa no tiene que ocuparse de verificar en qué localidad estás para decidir qué responder (como era habitual en PAWS), ya que ha sido el parser el que ha averiguado previamente si la antorcha está ahí o no.

Por tanto, resumamos otra vez el mecanismo:

El parser determina si la frase tiene sentido, y asocia a la frase una `accion` y un objeto sobre el que recae la acción (siempre que haya un objeto de ese nombre cerca).

Seguidamente la librería "avisa" a los objetos de que la acción tendrá lugar y si los objetos "no dicen nada", la librería llevará a cabo la acción por defecto.

¿Cómo "avisa" la librería a los objetos? Para que un objeto pueda ser "avisado" por la librería cuando una acción tiene lugar, el objeto debe proporcionar una rutina llamada `reaccionar_antes` (entonces la librería le "avisará" de todas las acciones que tienen lugar en la proximidad de ese objeto, aunque la acción no vaya referida a él), o bien una rutina llamada `antes` (entonces la librería le "avisará" sólo de las acciones que recaen sobre él, y no las que recaerían sobre otros objetos).

Olvidemos de momento la propiedad `reaccionar_antes` y centrémonos en la propiedad llamada `antes`. Esta propiedad se llama así como un recordatorio de que se activa *antes* de que la librería intente realizar la acción por defecto.

La propiedad `antes` es en realidad una "rutina incrustada en el objeto". ¿Recuerdas que antes hablábamos de "rutinas libres", como la rutina `Inicializar`? Ha llegado el momento de hablar de "rutinas incrustadas en los objetos".

Una rutina "incrustada en un objeto" es similar a una rutina libre en varios aspectos: se delimita mediante `[` y `]`, tiene una lista de variables locales y también una lista de instrucciones. Las principales diferencias son las siguientes:

- El nombre de la rutina no aparece después del `[`, sino antes de él.
- La rutina aparece dentro del objeto, como si fuera una propiedad de él (de hecho, en realidad es una propiedad del objeto).

Por ejemplo:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       saludo [;
                print "Hey, soy una antorcha!^";
              ],
has femenino;
```

Hemos añadido tres líneas a la antigua antorcha. En ellas, hemos programado una rutina incrustada, llamada `saludo`. Fíjate como el nombre de la rutina va antes del `[`. Después del `[` iría la lista de *variables locales*, que está vacía en este caso, y un punto y coma. A continuación, la lista de instrucciones que componen la rutina. En este caso sólo hay una instrucción que es `print`. Finalmente, el `]` indica el final de la rutina. Es necesario poner una coma detrás de este `]` porque todo lo anterior era en realidad una propiedad del objeto `antorcha`, y las propiedades van separadas por comas. El resto del objeto es como antes.

¿Qué conseguimos añadiéndole una "rutina incrustada"? En realidad nada. Si juegas de nuevo con esta antorcha modificada, verás que el juego es exactamente el mismo. La antorcha tiene una rutina nueva, pero esta rutina no es utilizada por el juego. Podrías utilizar la rutina si, en otra rutina, pones lo siguiente:

```
antorcha.saludo();
```

Esto es lo que se denomina una *llamada a rutina incrustada*. Como ves, las llamadas a rutinas incrustadas se

forman poniendo, en primer lugar, el objeto del que forman parte, después un punto y detrás el nombre de la rutina. Finalmente, son necesarios los paréntesis (a veces dentro de estos paréntesis puede ir información extra, que se le envía a la rutina, y es copiada en sus variables locales, pero no en este caso puesto que la rutina no tiene variables locales).

Cuando el juego llega a una instrucción como la anterior, buscará el objeto `antorcha` (no importa en qué lugar del juego esté, en este caso no es necesario que esté cerca del jugador), y buscará dentro del objeto `antorcha` una propiedad llamada `saludo`. Si esta propiedad no existiera, se produciría un error durante la ejecución del juego. Si existe, entonces se ejecutarán las instrucciones que hay dentro de la rutina (en nuestro ejemplo, el `print` que imprime el saludo), y cuando se encuentre una instrucción `return`, o bien cuando se llega al final de la rutina `]`, el juego continuará por la instrucción siguiente a la *llamada*.

En cierta forma, todo esto es similar a ejecutar un proceso PAWS, pero en este caso los procesos están incrustados dentro de los objetos, y no se identifican por un numero, sino por una pareja de nombres: el nombre del objeto y el nombre del proceso.

Bien, pero el caso es que nuestro juego no tiene ninguna instrucción que llame a `antorcha.saludo` y por eso el mensaje nunca será impreso.

Sin embargo, si en lugar de `saludo` hubieramos llamado a la rutina `antes`, el mensaje aparecería cada vez que se intentara hacer algo con la antorcha, ya que la librería se ocupa de llamar a `antorcha.antes` cada vez que detecta una acción con `uno=antorcha`. ¡Pruébalo! (tienes una versión modificada y lista para pruebas aqui

Si ejecutas esta versión, e intentas cosas con la antorcha, verás que el mensaje "Hey, soy una antorcha!^" aparece delante de cada respuesta de la librería. O sea, que la rutina `antes` es ejecutada *antes* que la acción de la librería, no obstante *no impide* que la acción por defecto tenga lugar.

Para impedir que la acción por defecto tenga lugar, la rutina `antes` tiene que ejecutar la instrucción `rtrue` (que viene a ser como el DONE del PAWS). `rtrue` significa "retorna VERDAD"; podemos imaginar que la librería le pregunta a la antorcha "¿verdad que ya te ocupas tú de realizar la acción?". Si la `antorcha` responde "VERDAD", entonces la librería ya no ejecutará la acción por defecto. Pero si la `antorcha` responde "MENTIRA" (`rfalse`), o no responde nada, la librería seguirá adelante con la acción por defecto.

¿Qué pasará por tanto si cambiamos la antorcha a esto otro?

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
                print "¡Dejame en paz!^";
                rtrue;
              ],
has femenino;
```

¡El jugador recibirá la respuesta "Dejame en paz" ante cada posible acción que intente sobre la antorcha! Y además la acción no será realizada. En particular, el jugador no podrá coger la antorcha. Todas las acciones fracasarán con el mismo mensaje.

Programando la antorcha (¡por fin!)

Cambiar las respuestas

Necesitamos un modo de discernir qué acción es la que está intentando el jugador, para que la respuesta de la antorcha no sea siempre la misma. Esto es muy sencillo, cambiemos la antorcha por esto:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger: print "La antorcha está incrustada en la pared.^";
                rtrue;
              ],
has femenino;
```

Ahora, el `print` y el `rtrue` sólo se ejecutarán si la acción era `Coger` (¡cuidado! Hablamos de acciones, no de verbos, recuerda). Cualquier otra acción es ignorada por la antorcha, y por tanto causará la acción por defecto.

Si queremos que la antorcha reaccione de la misma forma ante dos o más acciones diferentes, basta poner éstas separadas por comas, por ejemplo:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger, Girar: print "La antorcha está incrustada en la pared.^";
                rtrue;
              ],
has femenino;
```

Ahora el `print` y el `rtrue` se ejecutarán para las acciones `Coger` y `Girar`, las restantes acciones seguirán siendo tratadas por la librería.

Si queremos diferentes mensajes de respuesta ante diferentes acciones, basta ir repitiendo una estructura similar a la anterior, por ejemplo ¿qué pasa si el jugador pone SOPLA LA ANTORCHA? Intentémoslo (con el listado de acciones activado para ver qué acción se genera):

```
sopla antorcha
[ Acción Soplar con uno 30 (antorcha) ]
Tu soplido no produce ningún efecto.
```

¡Vaya! InformATE tenía prevista una acción `Soplar`, y la respuesta por defecto no parece del todo mala, pero cambiémosla por otra más apropiada:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger, Girar: print "La antorcha está incrustada en la pared.^";
                rtrue;
         Soplar: print "Iluso, el fuego que arde en esta antorcha no
                  es de este mundo.^";
                rtrue;
              ],
has femenino;
```

¡Acuerdate de poner el `rtrue`! (¿Qué pasaría si no lo pones?) La verdad es que repetir `rtrue` después de cada `print` es un poco pesado. Por eso Inform nos proporciona una abreviatura: el comando `print_ret`

imprime una cadena e inmediatamente retorna con `rtrue`. Además, imprime una "linea nueva" al final del texto, por lo que nos evita tener que poner el ^ en cada mensaje. Usando esta abreviatura, la cosa quedaría así:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger, Girar: print_ret "La antorcha está incrustada en la pared.";
         Soplar: print_ret "Iluso, el fuego que arde en esta antorcha no
                 es de este mundo.";
              ],
has femenino;
```

Aún es posible abreviar más. ¿No te suena esto de "imprimir, añadir salto de linea y retornar inmediatamente"? Todo ello podía abreviarse sin más que poner el mensaje entre comillas. Así:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger, Girar: "La antorcha está incrustada en la pared.";
         Soplar: "Iluso, el fuego que arde en esta antorcha no
                 es de este mundo.";
              ],
has femenino;
```

Esta forma es muy habitual, porque resulta muy cómoda. Sin embargo es un poco "rara" para el principiante, ya que en realidad se trata de una orden `print`, pero el `print` no se escribe. Además retorna inmediatamente (con `rtrue`), y eso tampoco se ve claramente en el código. Con todo, si sabes esto, no deberías tener problemas con su uso.

Como ves, añadir mensajes específicos para la antorcha resulta muy sencillo. ¡Añade tú mismo alguno! Por ejemplo, ¿tendrá InformATE previsto HUELE ANTORCHA? ¿COME ANTORCHA? ¿QUEMA ANTORCHA? ¿TOCA ANTORCHA? Pruebalo, verás que todos ellos están previstos, pero el mensaje de respuesta estándar puede no encajar muy bien. Averigua qué acciones genera cada una de las frases anteriores y modifica la antorcha para imprima mensajes más apropiados.

Haciendo que aparezca la puerta

Pero ahora pasemos a algo más serio. Hasta aqui, nos hemos limitado a variar los textos de respuesta de la antorcha, pero no hemos programado nada "realmente", en el sentido de que todas estas acciones no tienen efecto sobre el juego. Queremos que, al empujar la antorcha, aparezca una salida en `EscaleraCaracol2`. Recuerda que las salidas de una localidad son *propiedades* de esa localidad. Ahora tenemos un pequeño problema:

Una vez que el juego ha comenzado, no se pueden añadir o quitar propiedades a los objetos, pero sí que es posible cambiar el valor de cualquier propiedad de cualquier objeto.

Por tanto no podemos hacer que `EscaleraCaracol2` empieze sin su propiedad `al_o`, y más adelante en el juego de pronto adquiera esta propiedad. Sin embargo, lo que si podemos hacer es que `EscaleraCaracol2`, inicialmente tenga la propiedad `al_o`, pero esta propiedad tome el valor cero (que es lo mismo que decir que esa salida no lleva a ningún lugar). Más adelante, cuando se empuje la antorcha, cambiamos el valor de esta propiedad para que sea igual a `Mazmorra`.

Es decir, edita el juego y añade de nuevo la salida `al_o` en la escalera, pero en lugar de hacer que lleve a `Mazmorra`, haz que "lleve a 0", así:

```
Object  EscaleraCaracol2 "Escalera de caracol"
 with   descripcion "Los desgastados peldaños de piedra resbalan en
            ocasiones. A mitad de la escalera una antorcha en la
            pared impide que la oscuridad sea completa.",
        abajo EscaleraCaracol1,
        al_o 0,
        arriba AltoTorre,
 has    luz;
```

Cuando una salida "lleva a 0 (cero)", el jugador no podrá pasar por ella. Es como si no existiera. Pero el declararla de este modo nos permitirá cambiar su valor más adelante en el juego, mediante una instrucción como esta:

```
 EscaleraCaracol2.al_o = Mazmorra;
```

Que puedes leer así: "Cambia la propiedad `al_o` del objeto `EscaleraCaracol2`, y haz que sea igual a `Mazmorra`". Si quisieramos eliminar esa salida, basta hacerla igual a cero de nuevo:

```
 EscaleraCaracol2.al_o = 0;
```

¿Cuándo debe ejecutarse este código? Evidentemente cuando el jugador empuje o tire de la antorcha. Cuando empuja la conexión se crea, cuando tira, la conexión desaparece. Es decir:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger, Girar:  "La antorcha está incrustada en la pared.";
         Soplar:  "Iluso, el fuego que arde en esta antorcha no
                  es de este mundo.";
         Empujar: EscaleraCaracol2.al_o = Mazmorra;
                  "Al empujar la antorcha una porción de pared se abre
                   al oeste dando acceso a una estancia.";
         Tirar: EscaleraCaracol2.al_o = 0;
                "Al tirar la antorcha, la puerta secreta se cierra
                 de nuevo.";
              ],
has femenino;
```

¿Por qué he puesto la asignación antes del mensaje? ¿Daría igual hacerlo al revés?

Intenta probar si la cosa funciona. (torre5.zip tienes el código y el `z5` de esta versión). Empuja la antorcha, se abre la conexión, tira de ella, se cierra. ¡Bien! Empuja de nuevo, se abre otra vez. Y ahora que está abierta ¡empuja otra vez! ¿qué ocurre? Hum, esto no queda muy bien, el mensaje diciendo que la pared se mueve aparece de nuevo.

Deberíamos comprobar si la conexión existe antes de mostrar ese mensaje. Un poco más de programación:

```
Object antorcha "antorcha" EscaleraCaracol2
with   nombre 'antorcha',
       antes [;
         Coger, Girar:  "La antorcha está incrustada en la pared.";
         Soplar:  "Iluso, el fuego que arde en esta antorcha no
```

```
                es de este mundo.";
        Empujar: if (EscaleraCaracol2.al_o==0) {
                    EscaleraCaracol2.al_o = Mazmorra;
                    "Al empujar la antorcha una porción de pared se
                    abre al oeste dando acceso a una estancia.";
                }
                else "La antorcha no cede más.";
        Tirar: if (EscaleraCaracol2.al_o == Mazmorra) {
                   EscaleraCaracol2.al_o = 0;
                   "Al tirar la antorcha, la puerta secreta se cierra
                   de nuevo.";
                }
                else "La antorcha no cede más.";
             ],
has femenino;
```

Hemos usado aqui una estructura *condicional*, es decir, una forma de decirle a Inform que cierto código sólo debe ejecutarse si se cumple cierta condición. La estructura `if...else` tiene la siguiente sintaxis:

```
if (condicion) {
  ! Secuencia de instrucciones que deben ejecutarse
  ! si la condición se cumple
}
else {
  ! Secuencia de instrucciones que deben ejecutarse
  ! si la condición no se cumple
}
```

Observaciones:

- La secuencia de instrucciones a la que se hace referencia en esta sintaxis utiliza punto y coma (`;`) como delimitador de instrucciones.
- Si esta secuencia de instrucciones tiene una sola instrucción, entonces podemos no poner llaves `{`, `}` a su alrededor. Sin embargo, si tiene más de una instrucción es obligatorio poner esas llaves.
- La parte `else` es opcional. Si no la ponemos, entonces no se ejecutará nada en caso de que la condición no se cumpla.

¡Cuidado! Una pifia típica es olvidar las llaves cuando hay más de una instrucción. Considera el ejemplo siguiente:

```
if (talisman in jugador)
   AumentarFuerza();
   print "Sientes como tu fuerza aumenta.";
```

Parece que el programador pretendía aquí que la fuerza del jugador aumente si lleva un talismán consigo (la condición `talisman in jugador` sería cierta en este caso), y además mostrar un mensaje indicandole ésto al jugador. Sin embargo, ha olvidado poner llaves alrededor de estas instrucciones. ¿Qué crees que pasará? ¿Dará un mensaje de error Inform?

La respuesta es que no habrá mensaje de error, pero en cambio el juego no funcionará de la forma prevista. Seguramente lo verás más claro si elimino un par de espacios delante de `print` (para Inform es lo mismo que haya dos espacios o que haya doce):

```
  if (talisman in jugador)
     AumentarFuerza();
  print "Sientes como tu fuerza aumenta.";
```

¿Vés ahora mejor lo que ocurre? Al no poner llaves, Inform entiende que la "secuencia de instrucciones a ejecutar en caso de que la condición sea cierta", se compone de una sola instrucción: la llamada a la rutina `AumentarFuerza()`. El `print` que le sigue se considera que ya está "fuera del `if`", y por tanto no depende de la condición. Es decir, el mensaje "Sientes como tu fuerza aumenta" se imprimiría incluso si el jugador no lleva talismán (¡sin embargo su fuerza no aumentaría!)

Una vez explicado `if`, podemos ver con mayor claridad qué ocurre cuando el jugador empuja la antorcha. El código que se ejecutará entonces será el siguiente (un extracto de la rutina `antes` de la antorcha):

```
Empujar: if (EscaleraCaracol2.al_o==0) {
            EscaleraCaracol2.al_o = Mazmorra;
            "Al empujar la antorcha una porción de pared se
            abre al oeste dando acceso a una estancia.";
         }
         else "La antorcha no cede más.";
```

La condición que comprobamos es `EscaleraCaracol2.al_o==0`. El doble signo `==` significa que se intenta comprobar la igualdad, y no debe ser confundido con un solo `=` que sería una asignación como ya hemos visto. Por tanto la condición evaluada es "¿el valor de la propiedad `al_o` del objeto `EscaleraCaracol2` es igual a cero?" Si esta condición es cierta, se ejecutará lo que viene entre llaves a continuación. Si es falsa se ejecutará lo que va tras el `else`. Date cuenta de que si es cierta, esto implica que hacia el oeste la salida aún no existe, y por tanto lo que hacemos es crearla y emitir el mensaje. Y si es falsa, es que la salida hacia el oeste ya había sido creada, y por tanto emitimos un mensaje indicando que no ocurre nada más.

El código para el caso de la acción `Tirar` es totalmente análogo.

Condiciones

Inform puede comprobar muchos tipos de condiciones diferentes. Ya hemos visto uno de ellos: la igualdad. Las comparaciones numéricas tienen una sintaxis bastante lógica: `>` significa "mayor que", `<` es "menor que", `>=` significa "mayor o igual" y `<=` sería "menor o igual".

Además de estos casos sencillos, hay otras comprobaciones interesantes que necesitarás usar para programar juegos:

`~=`
: Desigualdad. El sigo `~` que aparece delante del igual se obtiene en la mayoría de los ordenadores pulsando AltGr y el 4. Si estás acostumbrado a programar en C, ten cuidado de no confundirte y usar `!=` ya que en Inform el carácter `!` es el comentario como ya hemos dicho.

`in`
: Estar dentro. El resultado de esta condición es `true` (verdad) si el objeto que nombras a la izquierda del `in` está dentro del que nombras a su derecha. Por ejemplo `antorcha in EscaleraCaracol2` es cierto, y sin embargo `antorcha in jugador` es falso (¡a menos que el jugador haya cogido la antorcha de alguna forma!)
  Ten en cuenta que en Inform los objetos pueden estar dentro de otros y estos a su vez dentro de otros. Si el jugador llevara consigo una caja dentro de la cual hay una moneda, sería cierto `caja in`

`jugador` y `moneda in caja`, pero no sería cierto `moneda in jugador` (la condición `in` se limita a comprobar la inclusión en un solo nivel).

has

Tener un atributo. El resultado de esta condición es `cierto` si el objeto que nombras a la izquierda de `has` tiene activado el atributo que nombras a su derecha. Por ejemplo `antorcha has femenino` sería cierto (puesto que en la declaración del objeto `antorcha` le hemos puesto el atributo femenino), o `EscaleraCaracol2 has luz` también sería cierto (pues esta localidad tiene el atributo `luz`). En cambio, `antorcha has luz` es falso pues no le hemos dado este atributo a la antorcha (pensandolo bien, parece que deberíamos haberselo dado, ¿no? De todas formas en este juego es intrascendente porque la antorcha nunca se mueve de su localidad)
Observa que `has` se limita a evaluar si el objeto dado tiene ese atributo, pero no tiene inteligencia alguna para hacer deducciones. Por ejemplo, si llevas una linterna que tiene el atributo `luz` y con ella entras en una cueva que no tiene el atributo `luz`, será cierto `linterna has luz` pero falso que `cueva has luz`. Es decir, `has` no es capaz de deducir que hay luz en la cueva (suministrada por la linterna). Para comprobar si hay luz en una localidad deberías comprobar si la localidad "`has luz`" o si alguno de los objetos que tiene dentro "`has luz`". Por suerte no tienes que molestarte en comprobar esto. La librería lo hace automáticamente cada turno (de hecho hace algo mucho más complejo, pues tiene en cuenta si los objetos están dentro de otros. Así, si llevas la linterna dentro de una caja opaca, la librería deduciría que no hay luz visible, pero si la llevas dentro de una caja transparente sí la habría).

hasnt

Lo contrario de `has`. La condición `objeto hasnt atributo` es cierta si el objeto tiene ese atributo desactivado. Por tanto, `Cueva hasnt luz` será cierto si la cueva no tiene luz.

Hay más condiciones, pero estas son las más importantes y las más usadas. Puedes combinar varias condiciones en una sola. Por ejemplo, si quieres que algo se ejecute sólo si el jugador lleva un talismán, y *además* el talismán tiene `luz`, puedes combinar ambas condiciones usando el operador &&, en la forma siguiente:

```
if ((talisman in jugador) && (talisman has luz)) ...
```

Observa cómo hay que encerrar cada condición entre paréntesis, y colocar el operador && entre ambas.

Si lo que quieres es ejecutar algo cuando es cierta una condición cualquiera entre varias, por ejemplo, si el jugador lleva el talismán o si lleva el anillo, entonces el operador es `||` (el símbolo `|` se obtiene pulsando AltGr y 1):

```
if ((talisman in jugador) || (anillo in jugador)) ...
```

Puedes poner tantas condiciones como quieras combinadas con estos operadores, e incluso mezclar operadores. Piensa por ejemplo cómo harías para escribir una condición que sea "si el jugador lleva el talismán con luz, o bien si lleva el anillo puesto" (para verificar si lleva el anillo puesto debes mirar si `anillo has puesto`, además de comprobar que está en el jugador).

Inform tiene también una abreviatura muy cómoda para el caso de que estés comprobando varias igualdades alternativas. Por ejemplo, imagina que quieres emitir un mensaje si el jugador está en la localidad llamada `Mazmorra`, o en la localidad `Pasadizo` o en la localidad `Sotano`. En principio puede hacerse usando el operador `||` para diferentes comparaciones con la variable `localizacion`, así:

```
if ((localizacion==Mazmorra) || (localizacion==Pasadizo)
    || (localizacion==Sotano))  print "¡Qué frio!^"
```

Pero también es posible usar la siguiente forma abreviada:

```
if (localizacion == Mazmorra or Pasadizo or Sotano)
   print "¡Qué frio!^";
```

Esta abreviatura (`or`) sólo puede aplicarse para comparaciones de igualdad, en las que el elemento a la izquierda de la comparación es siempre el mismo, y queremos compararlo contra diferentes valores.

Parte II: Empieza lo interesante

Una vez vistos los rudimentos de programación bajo Inform, en esta segunda entrega nos centraremos en los elementos de que están hechos los objetos: propiedades y atributos. InformATE tiene docenas de propiedades y atributos, y el principiante suele verse abrumado por tal cantidad de información. No te preocupes, en este cursillo no trataremos de verlos todos, pero sí los más importantes, a medida que los vayamos necesitando. Y como verás, sólo con lo que aquí se explica ya podrás construir juegos suficientemente complejos e interesantes.

Mejorando detalles estéticos

Más sobre atributos

Un atributo es una característica que un objeto dado puede tener o bien no tener. Por ejemplo, la luz de una habitación. Son características del tipo SI/NO sin posibilidad de valores intermedios. Si necesitas que en tu juego la luz sea algo más elaborado (por ejemplo: mucha luz, luz normal, penumbra, semioscuridad, oscuridad total), entonces no podrás usar un atributo para representar la luz. Para representar características que pueden tomar varios valores se deben usar "propiedades" (que veremos en una sección posterior).

Inform tiene predefinidos un buen montón de atributos. Cuando creas un objeto puedes "activar" cualquiera de ellos. Los que no actives explícitamente se consideran "desactivados". Ya hemos visto cómo se hace esto en los ejemplos anteriores, ya que todas las localidades tenían el atributo `luz`, y la antorcha tenía el atributo `femenino`. Un objeto puede tener varios atributos activos a la vez, así por ejemplo la antorcha, además de `femenino` podría tener `luz`. Para lograr esto basta poner todos los atributos que se quieran separados por espacios. Por ejemplo:

```
Object antorcha "antorcha"
with nombre 'antorcha' 'tea',
 ....
has femenino luz;
```

También puedes activar o desactivar un atributo en cualquier momento del juego. Por ejemplo, una localidad podría comenzar sin el atributo `luz` (y el jugador no podría ver nada al entrar en ella), pero hacer que, si el jugador pulsa un interruptor localizado en otro lugar, entonces se active la luz de esta habitación. Esto es muy

sencillo, ya que para poner un atributo a un objeto basta la instrucción:

```
give objeto atributo;
```

que se leería "dale al objeto este atributo". De modo que para activar la luz de una localidad llamada `Tunel` la cosa sería tan simple como: `give Tunel luz;` La próxima vez que el jugador entre en esa localidad, ya podrá ver (y si el jugador estaba en esa localidad en el instante en que esta instrucción es ejecutada, al final de este turno "se hará la luz" y se le mostrará al jugador la descripción del lugar).

Para desactivar un atributo (por ejemplo, para apagar la luz de una localidad) la sintaxis es un poco rara:

```
give objeto ~atributo;
```

que se leería "dale al objeto este atributo desactivado". El signo ~ delante de un atributo indica su desactivación. Así pues, para apagar las luces del tunel, por ejemplo sería: `give Tunel ~luz;`

El hecho de activar un atributo en un objeto, tiene efectos sobre el juego. En el ejemplo de la `luz` la cosa está clara. Pero hay otros muchos atributos y cada uno tiene su efecto sobre el juego. Por ejemplo, existe el atributo `abierto` que indica cuando un recipiente está abierto, y según el valor de este atributo el juego te dejará ver o no lo que hay dentro del recipiente.

Iremos viendo algunos atributos y sus efectos sobre el juego a medida que los vayamos necesitando.

Mejorando la antorcha

Sólo nos queda un detalle por mejorar en la antorcha. Observa de nuevo la descripción de la localidad:

```
Escalera de caracol
Los desgastados peldaños de piedra resbalan en ocasiones.
A mitad de la escalera una antorcha en la pared impide
que la oscuridad sea completa.

Puedes ver una antorcha.
```

Vaya. La antorcha es mencionada dos veces. Esto se debe a que Inform primero imprime la descripción de la localidad, y seguidamente crea una lista de todos los objetos que hay en esa localidad, que encabeza con "Puedes ver..."

Por suerte (o por desgracia, según se mire) el programador tiene muchas formas de afectar este mecanismo, y puede lograr entre otros los siguientes efectos:

- Que Inform no mencione un objeto en la lista "Puedes ver...", aunque se halle realmente en esa localidad. Esto puede sernos útil para la antorcha.
- Que un objeto concreto tenga una "linea para el solo". Por ejemplo, imagina que en la localidad hay un caballo, y una moneda que el jugador ha dejado en el suelo. Inform añadiría al final de la descripción de la localidad la frase: "Puedes ver un caballo y una moneda".

  Parece que el caballo requeriría más protagonismo. Es fácil asignarle al caballo una línea para él solo, con el texto que el programador desee, de forma que el resultado sea este otro: "Un caballo de aspecto imponente te mira con desconfianza. Puedes ver también una moneda."

- Que un objeto tenga diferente descripción según esté abierto o cerrado, o según esté encendido o

apagado. Así, en lugar de "Puedes ver una caja", Inform puede producir la frase "Una caja abierta yace en el suelo." o "Hay una caja cerrada en el suelo", según el valor del atributo `abierto` de ese objeto.

- Que un objeto tenga diferente descripción la primera vez (hasta que el jugador lo coge). Así, por ejemplo, podemos hacer que un objeto `Seta` se mencione con la frase: "Del suelo emerge una seta de tallo estrecho", pero tan pronto como el jugador la coja y la suelte de nuevo, pasa a la forma por defecto "Puedes ver una seta.".

Como ves, las posibilidades son muchas. Para lograr uno u otro efecto, debemos hacer uso de atributos y propiedades. Vamos a centrarnos de momento en nuestro problema, que es hacer que no se mencione la antorcha en la linea "Puedes ver..." que es el caso más sencillo.

Para lograr esto, basta que la antorcha tenga el atributo `oculto`. Y puesto que queremos que este atributo esté activo desde el principio del juego, lo mejor es dárselo ya en la propia declaración de la antorcha.

Vete por tanto a la línea final de la antocha, que tenía `has femenino` y cambiala por:

```
has femenino oculto;
```

El nombre de este atributo es un poco engañoso. Quizás debería haberse llamado `no_mencionar`, o algo así, ya que realmente lo único que logras con este atributo es que Inform no mencione ese objeto cuando describa la habitación. Sin embargo, el objeto no está realmente oculto, ya que el jugador podría cogerlo o realizar cualquier otra acción sobre él.

Si compilas y ejecutas esta versión, verás que en la localidad de la antorcha ya no se menciona esta (salvo, claro está, en la propia descripción de la localidad). Por tanto hemos logrado el objetivo.

Otro atributo con efectos similares es el `escenario`. Cuando un objeto tiene este atributo, tampoco es mencionado en la localidad. Además, por defecto la librería no dejará efectuar ninguna acción sobre ese objeto que implique moverlo. Por ejemplo, empuja, tira, coge, mueve, etc... todas fracasarán. En cambio sí permite examinarlo. De este modo te ahorras tener que programar tú mismo en la rutina `antes` todas las respuestas necesarias para evitar estas acciones.

Podríamos haber puesto a la antorcha el atributo `escenario` y esto no afectaría a su correcto funcionamiento ante `Empujar` o `Tirar`, ya que la antorcha tiene su propio código para manejar estas acciones, y el código del objeto siempre tiene prioridad sobre la acción por defecto de la librería. Sólo cuando la antorcha no contemple una acción, se ejecutará el caso por defecto para la librería. De hecho, sería más prudente hacer que la antorcha sea `escenario`, pues de este modo, si se nos ha olvidado programar alguna respuesta por defecto ante alguna acción que implique mover la antorcha, la librería tendrá una por defecto una que impedirá ese movimiento. Por ejemplo, imagina que hubieramos olvidado programar la acción `Coger` de la antorcha. Si la antorcha es `escenario`, la librería impedirá esta acción. Sin embargo, si no es `escenario`, ¡el jugador podría llevarse la antorcha a otra localidad! Y lo que es peor, una vez que se la ha llevado, si la empuja o tira de ella, ¡¡el mecanismo de la puerta de la mazmorra seguiría funcionando!!

En resumen, si quieres asegurarte de que un objeto nunca va a poder ser llevado de su localidad de origen, ponle el atributo `escenario`.

Por otra parte, `escenario` implica `oculto`, es decir, que el objeto no es mencionado en la descripción de la localidad. Si quieres evitar que un objeto pueda moverse, pero quieres que Inform lo mencione normalmente en la linea "Puedes ver..." debes darle el atributo `estatico`.

Descripciones variables

Ahora quedaría bonito que la descripción de la localidad hiciera referencia a la entrada a la mazmorra (si está abierta, claro). Esto es mucho más fácil de lo que puede parecer a primera vista, ya que Inform permite que la descripción de una localidad sea una rutina embebida, en lugar de un texto estático.

Es decir, podemos poner, en lugar de una cadena entre comillas como aquí:

```
 descripcion "Bla bla bla",
```

Una rutina como aquí:

```
 descripcion [;
      print "Bla bla bla";
    ],
```

En este caso la diferencia no es mucha, ya que el resultado que verá el jugador será exactamente el mismo. Sin embargo para el programador es todo un mundo de diferencia, porque si la descripción es una rutina ¡puede usar `if` para generar diferentes descripciones, según se cumplan o no ciertas condiciones!

Haciendo que una propiedad sea una rutina, podemos lograr que produzca diferentes resultados en diferentes instantes del juego. Esta idea no sólo vale para la descripción. Como veremos, en Inform prácticamente cualquier propiedad puede ser, bien un valor fijo, o bien una rutina.

Aplicando esta idea a la localidad `EscaleraCaracol2`, podemos hacer que la descripción añada una observación sobre la entrada a la mazmorra o que calle, según el valor de la salida `al_o`. La nueva programación de esta localidad sería:

```
Object  EscaleraCaracol2 "Escalera de caracol"
 with   descripcion [;
            print "Los desgastados peldaños de piedra resbalan en
              ocasiones. A mitad de la escalera una antorcha en la
              pared impide que la oscuridad sea completa.^";
            if (EscaleraCaracol2.al_o==Mazmorra)
               print "^Parte de la pared oeste ha deslizado mostrando
                   una entrada.^";
        ],
        abajo EscaleraCaracol1,
        al_o 0,
        arriba AltoTorre,
 has    luz;
```

El código con esta mejora (y con la antorcha `oculta`) lo tienes aqui, junto con una versión `z5`, lista para que la ejecutes y pruebes.

Rompecabezas clásicos

El rompecabezas más viejo y más usado es aquel en el que un objeto está oculto por otro, y sólo se revela al examinar el segundo. Tenemos un par de rompecabezas así en nuestra aventura. En la mazmorra hay un esqueleto que al ser examinado revela un cuchillo, y en el dormitorio hay unas camas que al ser examinadas revelan unas fundas. También vamos a esconder un trozo de carbón en la chimenea, aunque no sirva para nada en el juego.

Hemos visto que InformATE tiene un atributo engañosamente llamado `oculto`, que parece invitar a ser usado para esto. Sin embargo hemos visto también que `oculto` simplemente hace que la librería no mencione el objeto, pero no impide que el jugador se refiera a él.

Es decir, si pusieramos en la mazmorra un objeto `Esqueleto` y un objeto `cuchillo`, y diéramos a éste último el atributo `oculto`, Inform le diría al jugador al entrar en la localidad: "Puedes ver un esqueleto" (silenciando toda mención al cuchillo). Sin embargo, el jugador podría pese a todo escribir "COGE CUCHILLO" o "EXAMINA CUCHILLO" y la librería le permitiría la acción.

La única forma de impedir que el jugador pueda referirse al cuchillo es ¡no poner un cuchillo en esa localidad! Si un objeto está en una localidad, el jugador siempre podrá referirse a él como blanco de sus acciones.

Por tanto la forma típica de programar esto consiste en tener programado el objeto cuchillo, pero no ponerlo dentro de ninguna de las localidades del juego. De este modo, cualquier intento de manipular el cuchillo por parte del jugador, causará el mensaje de error "No veo eso por aqui".

Programaremos después el esqueleto de forma que, tan pronto como el jugador lo examine, el cuchillo será movido a la `Mazmorra`. Así, a partir de este punto el cuchillo se convertirá en un objeto más del juego, que podrá ser manipulado o no, según sus atributos y sus propias reglas programadas en su rutina `antes`.

Para crear un objeto que no está en ninguna localidad, basta no poner en su cabecera en qué localidad se halla. Recuerda que éste era el último parámetro que se daba en la cabecera (echale un vistazo al objeto `antorcha`). Si el objeto cuchillo empieza así:

```
Object cuchillo "cuchillo"
```

al no poner en qué localidad se halla, resultará que no está en ninguna. De todas formas, es buena idea crear una localidad ficticia, que yo suelo llamar "Limbo", y meter en ella todos los objetos que no aparecen inicialmente en el juego. Cuando el objeto en cuestión deba aparecer, bastará moverlo desde el "Limbo" a la localidad apropiada. Y a la inversa, para hacer desaparecer un objeto del mundo del juego, bastará moverlo al "Limbo" (si no usaramos Limbo, la forma de hacer desaparecer un objeto sería usar `remove objeto`, lo que en realidad lo saca de todas las localidades).

Esqueleto y cuchillo

Programemos de momento un cuchillo mínimo, que conste simplemente de su nombre y una descripción, y hagamos que inicialmente se halle en la localidad ficticia llamada "Limbo". Debemos crear también esta localidad, claro. Esto puedes escribirlo en cualquier lugar del código, después de incluir Acciones. Un buen sitio puede ser justo antes de la rutina `Inicializar`, para tener juntos todos los objetos de limbo y encontrarlos fácilmente, para cuando haya que cambiar algo en su código.

```
Object Limbo "Limbo"
```

```
with descripcion "Eh. ¿cómo has llegado aquí? Malditos
  betatesters...",
has luz;

Object cuchillo "pequeño cuchillo" Limbo
with nombre 'pequeno' 'cuchillo' 'punal' 'hoja' 'cuchilla',
     descripcion "Un pequeño cuchillo, cubierto de herrumbre";
```

Observa como he evitado el uso de la eñe en las palabras de vocabulario (las que figuran en la propiedad `nombre`). Esto no impide que el jugador pueda escribir "EXAMINA PEQUEÑO PUÑAL", por ejemplo. Si el jugador usa eñes, Inform las convertirá en enes antes de comenzar el parsing. Sin embargo, si hubieramos puesto 'puñal' en la lista de nombres de este objeto, estaríamos discriminando a los jugadores que intentan jugar desde un ordenador sin eñes, ya que nunca podrían escribir esa palabra. Lo mismo vale para los acentos. No uses acentos en las palabras de vocabulario si quieres lograr la máxima compatibilidad y difusión de tu juego.

Ahora debemos programar el esqueleto, y necesitamos que al ser examinado haga aparecer el cuchillo. Sin embargo hay que estar al tanto de un importante detalle. El cuchillo debe aparecer sólo *la primera vez* que se examina el esqueleto, y no cada vez que se le examine. Por tanto necesitamos llevar de algún modo el registro de si el esqueleto ya ha sido examinado o no.

Para estos menesteres, todos los objetos del juego tienen un atributo llamado `general`, que por defecto está desactivado en todos ellos. La librería nunca activa este atributo, ni lo consulta para nada. Por tanto, queda libre para que el programador lo use para sus propias necesidades. El uso más frecuente de este atributo es como indicador de "puzzle resuelto", de modo que tan pronto como el jugador resuelve algo (encuentra el cuchillo, por ejemplo), el objeto que causaba el puzzle (el esqueleto) recibe el atributo `general`, como indicador de "esto ya está". El código que hace aparecer el cuchillo, por tanto, sólo lo hará si el esqueleto no tiene ya activado el atributo `general`.

Vamos pues con la programación del esqueleto. El siguiente código conviene que lo pongas después del de la `Mazmorra`, para tener los objetos cerca de las localidades en que aparecen.

```
Object Esqueleto "esqueleto" Mazmorra
 with   nombre 'esqueleto' 'humano' 'muerto' 'cadaver',
        descripcion [;
            print "Los huesos amarillentos, las cuencas vacías";
            if (Esqueleto hasnt general)
            {
                move cuchillo to Mazmorra;
                print ". Junto a él ves un pequeño cuchillo";
                give Esqueleto general;
            }
            ".";
        ],
 has    escenario;
```

El código que hace aparecer el cuchillo lo hemos puesto dentro de la propia descripción del esqueleto, haciendo uso de la posibilidad de que las descripciones sean rutinas. Antes de hacer aparecer el cuchillo, verificamos si el esqueleto tiene el atributo `general`, pues esto indicaría que el cuchillo ya ha sido encontrado y no debe aparecer de nuevo. La condición para comprobar que un objeto *no tenga* un atributo, es `hasnt` (opuesto de `has`).

Si el esqueleto *no tiene* `general`, movemos el cuchillo a la mazmorra. La instrucción Inform para esto es `move obj1 to obj2`. El `obj2` no tiene por qué ser necesariamente una localidad. Podría ser un

recipiente, o incluso el propio jugador (en cuyo caso el `obj1` pasaría directamente a su inventario). Seguidamente imprimimos un mensaje para informar al jugador de su descubrimiento y finalmente activamos el atributo `general` del esqueleto, para señalar que el puzzle ha sido resuelto y así evitar que el cuchillo aparezca al examinarlo por segunda vez.

La línea que pone simplemente `"."`, como ves, está fuera del `if`, y por tanto se ejecutará siempre, independientemente de la condición. Se trata de uno de esos `print` implícitos, que imprimirá el punto final de la frase, un retorno de carro, y finalizará la función retornando `true`. Observa cómo hemos omitido el punto final en los `print` anteriores. También podríamos haber puesto el punto final en ambos `print`, y omitir el que va fuera del `if`, pero en este caso convendría sustituir este último punto por un `rtrue`, para asegurarse de que nuestra función devuelve `true`. Si bien en este caso concreto da lo mismo lo que la función retorne, es buena costumbre retornar `true` cuando nuestra rutina ha mostrado algún mensaje que debe sustituir a los defectos de la librería.

En Inform siempre hay muchas formas diferentes de lograr un mismo resultado. Por ejemplo, podríamos haber escrito el código que hace aparecer el cuchillo como parte de la rutina `antes` del esqueleto (ante la acción `Examinar`), en lugar de ponerlo en su `descripcion`. En este caso sí se haría imprescindible retornar `true`, pues si no lo hicieramos, después de nuestros mensajes la librería imprimiría los suyos (que consistirían en la descripción del esqueleto, si éste tiene una propiedad `descripcion`, o en el mensaje "No observas nada especial en el esqueleto" si no la tiene).

Convendría ahora pararse a testear si nuestro puzzle funciona correctamente, y de paso intentar unas cuantas acciones previsibles sobre el esqueleto o el cuchillo (cogerlo, moverlo,...) para asegurarnos de que las respuestas por defecto de la librería son adecuadas en todos los casos, o para sustituir las que no lo sean mediante una rutina `antes` en estos objetos. Aqui tienes una versión del juego que incorpora el `Limbo`, el `cuchillo` y el `esqueleto`, en código fuente y en versión `z5` lista para probar.

Verifica (mediante el comando `xlista`) que el cuchillo inicialmente está en el limbo, y comprueba (de nuevo con `xlista`) como se mueve a la mazmorra tras examinar el esqueleto.

Otro verbo de depuración que puede interesarte es el verbo `CAMBIOS`. Si pones esto en el juego, a partir de ese instante Inform te avisará cada vez que un objeto se mueva, un atributo se ponga o quite mediante `give`, o una propiedad cambie de valor mediante una asignación.

Vamos a probarlo. Tal vez te sorprenda ver la gran cantidad de atributos que cambian cada vez que efectúas algunas acciones. Sigue mis pasos: carga el juego, escribe `IRDONDE ANTORCHA` (para teleportarnos a la escalera de caracol) y seguidamente `CAMBIOS`. Agarrate fuerte a la silla y pon:

```
>empuja antorcha
[Giving antorcha ~nombreusado]
[Giving antorcha nombreusado]
[Setting Escalera de caracol.al_o to 28]
Al empujar la antorcha una porción de pared se abre al oeste
dando acceso a una estancia.
```

Parece que la librería usa un atributo llamado `nombreusado`, que es quitado y puesto de nuevo en la antorcha. Puedes olvidarte de este atributo, el parser lo usa continuamente en sus deducciones de qué objeto has nombrado, por lo que aparecerá en todos los turnos.

Vemos además, que ha cambiado el valor de la propiedad `al_o` de la escalera de caracol, a la que ha dado el valor 28. Este 28, evidentemente, debe ser el número interno de la localidad

`Mazmorra` (puedes verificarlo con `xlista`).

Prosigamos, y caminemos hacia el oeste:

```
>o
[Giving oeste nombreusado]
[Moving ti mismo to Mazmorra]

Mazmorra

Una silenciosa estancia débilmente alumbrada por los rayos
de luna que se filtran a través de un pequeño ventanuco. El suelo
está lleno de paja, colgando de unos grilletes en la pared
observas un esqueleto humano.
[Giving Mazmorra visitado/visitada]
```

Ignoremos la línea que hace referencia al atributo `nombreusado`. La línea "[Moving ti mismo to Mazmorra]" indica que el objeto `jugador` ha sido movido al objeto `Mazmorra`. La línea "[Giving Mazmorra visitado/visitada]" indica que se ha activado el atributo `visitado` del objeto `Mazmorra`. Este atributo tiene un "alias" (otro nombre) que es `visitada`. Su función, como es evidente, es indicar si esa localidad ya ha sido visitada con anterioridad o no. La librería puede cambiar algunas cosas de la descripción en caso de que el jugador ya haya estado allí (por ejemplo, puede no escribir descripción en absoluto y limitarse a poner el título de la localidad. Como ya vimos, esto depende también del valor de la variable `modomirar`).

Finalmente, veamos como aparece el cuchillo:

```
>x esqueleto
[Giving oeste ~nombreusado]
[Giving esqueleto nombreusado]
Los huesos amarillentos, las cuencas vacías[Moving pequeño
cuchillo to Mazmorra]
. Junto a él ves un pequeño cuchillo[Giving esqueleto
general]
.
```

Además de los mensajes acerca de `nombreusado`, vemos también que, tras la descripción del esqueleto, se nos informa de que el cuchillo ha sido movido a la mazmorra. Después aparece el mensaje para el jugador "Junto a él ves...", tras el cual Inform nos avisa de que se ha activado el atributo `general` del esqueleto. Finalmente, el punto final (aparece aqui en línea aparte porque los avisos entre corchetes añaden siempre un salto de línea).

Como ya hemos dicho, hay muchas formas de programar una misma idea en InformATE. Por ejemplo, para verificar si el cuchillo ya había sido encontrado, podríamos haber mirado si estaba o no estaba en el limbo, en lugar de usar el atributo `general` del esqueleto. En este enfoque (más parecido al que seguiría un programador de PAWS) bastaría algo como:

```
        descripcion [;
            print "Los huesos amarillentos, las cuencas vacías";
            if (cuchillo in Limbo)
            {
                move cuchillo to Mazmorra;
                print ". Junto a él ves un pequeño cuchillo";
            }
            ".";
        ],
```

Este mecanismo, de hecho, parece más simple. Pero imagina que decidimos permitirle al jugador que lance el cuchillo por la ventana, y como "premio" a su inteligencia, hacemos desaparecer el cuchillo del juego moviendolo de nuevo al Limbo. En este caso, con la programación anterior, ¡el jugador haría reaparecer el cuchillo examinando el esqueleto de nuevo! La cosa podría solucionarse teniendo dos Limbos diferentes: un Limbo1 para los objetos que aun no han aparecido en el juego y un Limbo2 para los que han sido eliminados.

Examinar en Inform

La librería InformATE tiene varias acciones relacionadas con la idea de examinar un objeto. Se trata de `Examinar`, `BuscarEn` (que puede entenderse como "registrar o mirar dentro de un objeto") y `MirarDebajo`. Los verbos que el jugador puede usar para dar lugar a estas acciones son muy variados:

Examinar

Esta acción es causada por cualquiera de los verbos siguientes: "EXAMINA OBJETO", "X OBJETO", "EX OBJETO", "MIRA OBJETO", "MIRA HACIA OBJETO", "DESCRIBE OBJETO", "INSPECCIONA OBJETO", "OBSERVA OBJETO" y "LEE OBJETO".

Observa que "LEE OBJETO" causa la acción `Examinar`. No hay acción `Leer` en InformATE, aunque esto puede cambiarse si lo necesitas.

BuscarEn

Esta acción es causada por cualquiera de las formas siguientes: "MIRA EN OBJETO", "MIRA DENTRO DE OBJETO", "MIRA SOBRE OBJETO", "MIRA A TRAVES DE OBJETO", "MIRA POR OBJETO", "BUSCA EN OBJETO", "REGISTRA OBJETO" y "REGISTRA EN OBJETO".

Observa que InformATE no entiende por ejemplo "EXAMINA EN OBJETO". Esto es lógico, ya que esa frase es incorrecta en español, pero puede despistar a algunos jugadores que piensen erróneamente que "MIRA" equivale a "EXAMINA" en todos los contextos. Recuerda que en InformATE el significado de un verbo depende de la frase en que aparece, y no solo de la palabra que lo representa.

MirarDebajo

Esta acción es causada por las formas: "MIRA BAJO OBJETO" y "MIRA DEBAJO DE OBJETO".

De nuevo, "EXAMINA DEBAJO DE OBJETO" no sería una forma válida para InformATE.

Es de destacar que InformATE no incorpora verbos ni acciones para la idea de "mirar detrás de algo". Incluso la idea de "mirar a través de algo" (como una ventana) o "Mirar por algo" (como mirar por un telescopio), aunque sí están contempladas, desembocan en la acción `BuscarEn` en lugar de tener sus propias acciones. Conviene estar sobreaviso de todos estos detalles.

En nuestro esqueleto, el cuchillo aparecerá si la acción generada por el jugador ha sido `Examinar` (ya que es esta acción la que da lugar a la ejecución de la rutina `descripcion` del esqueleto). Sin embargo, si el jugador genera `BuscarEn` o `MirarDebajo`, al no haber contemplado en el esqueleto estas acciones, la respuesta del juego sería: "No encuentras nada interesante" y "No ves nada interesante", respectivamente. Esto puede considerarse un bug del juego, ya que si lo primero que pone el jugador, en lugar de "EXAMINA

ESQUELETO" es "REGISTRA ESQUELETO", el cuchillo no aparecerá.

Bien, es cierto que el cuchillo está "junto al esqueleto", y por tanto al registrarlo no debería aparecer, pero sin embargo considero "juego sucio" obligar al jugador a poner "EXAMINA ESQUELETO" para descubrir el cuchillo. Deberíamos ser tolerantes y admitir cualquier intento de búsqueda en el esqueleto.

Por tanto, debemos escribir una rutina `antes` para nuestro esqueleto, que se haga cargo de las acciones `BuscarEn` y `MirarDebajo`, y que cambie los mensajes por defecto para este caso. Por ejemplo:

```
  antes [;
    BuscarEn, MirarDebajo: if (Esqueleto hasnt general)
              {
                 move cuchillo to Mazmorra;
                 give Esqueleto general;
                 "Al acercarte al esqueleto ves a su lado un pequeño
                  cuchillo.";
              }
              else "Este desgraciado no llevaba nada más.";
   ],
```

Fíjate cómo trato ambas acciones de la misma forma, poniéndolas separadas por comas antes de los dos puntos. Esta sintaxis indica que si la acción es `BuscarEn` o `MirarDebajo`, debe ejecutarse ese código. El código en cuestión es similar al que había escrito en la descripción del esqueleto. Pero he movido la línea `give esqueleto general` delante del mensaje (¿sabes por qué?).

Piensa cuáles serían las respuestas del juego si el jugador primero EXAMINA el esqueleto, y después lo REGISTRA. Piensa también cómo serían si hace estas acciones en orden inverso. Comprueba lo que ocurre realmente jugando el juego que tienes aqui

La chimenea y el carbón

Una vez que hemos visto la idea clave de usar `general` como indicador de puzzle resuelto, el problema de encontrar un carbón dentro de la chimenea se resuelve de forma análoga al del cuchillo:

```
Object carbon "trozo de carbón" Limbo
with nombre 'trozo' 'carbon',
     descripcion "Un trozo de negro carbón que parece haber
                  sobrevivido al fuego.";

Object chimenea "chimenea" PuertaPrincipal
with nombre 'chimenea' 'hogar',
     descripcion [;
          print "Hace mucho tiempo que no arde fuego alguno
              en esta vieja chimenea";
          if (chimenea hasnt general) {
             move carbon to PuertaPrincipal;
             print ". Un trozo de carbón es todo lo que queda
                 del antiguo hogar";
             give chimenea general;
          }
          ".";
     ],
has femenino escenario;
```

Observa un detalle en la propiedad `nombre` del trozo de carbón. No hemos incluido la palabra "de" en esta

propiedad ¿cómo es posible que el parser entienda entonces "coge trozo de carbón"?

El parser de InformATE está preparado para ignorar la preposición "de" si esta aparece entre dos palabras que se refieren al mismo objeto. Así, en "COGE TROZO DE CARBON" el "DE" es ignorado por el parser, puesto que "TROZO" y "CARBON" se refieren a un mismo objeto. La acción resultante será `Coger` con `uno==carbon`.

En cambio, en una orden como "COGE CARBON DE CHIMENEA", la preposición "DE" no es ignorada, pues "CARBON" y "CHIMENEA" no son palabras que estén en el `nombre` de un mismo objeto. En este caso "DE" se entiende como una preposición que separa dos objetos diferentes, y por tanto para que el parser comprenda esta orden debe existir una gramática que le diga que la sintaxis "COGE `<cosa>` DE `<otra cosa>`" es correcta. (De hecho, la gramática contempla esta forma y generaría la acción `Sacar` con `uno==carbon` y `otro==chimenea`)

La cama y la funda

Te dejaré como ejercicio, amable lector, que codifiques tú mismo este último puzzle. Cuando el jugador examine la cama por primera vez, debe aparecer el mensaje "Una funda de tela cubre la cama", y en los sucesivos exámenes de la cama "Sólo restos de paja cubren la cama".

Naturalmente, la funda no debe ser accesible hasta que el jugador haya examinado la cama. En cuanto a la paja, la versión fácil del ejercicio no hace aparecer ninguna paja, de modo que si el jugador escribe "EXAMINA PAJA" simplemente recibirá un "No veo eso por aquí". ¿Te atreves con una versión difícil en la que realmente aparezca un objeto `paja` al examinar la cama por segunda vez?

Por el momento, la funda puede ser cogida por el jugador. En la próxima entrega haremos que no pueda cogerla hasta que haya cortado las ataduras que la sujetan a la cama.

Aquí tienes la versión del juego que incluye la chimenea con su carbón, y la cama con su funda y la paja que aparece al mirar una segunda vez.

Rompecabezas un poco más complicados

Vamos a programar el rompecabezas de la funda. Se trata de que la orden COGE FUNDA fracase, a menos que el jugador antes haya cortado las correas con el cuchillo. ¿Cómo lograr esto?

Para empezar habrá que añadir un nuevo objeto: las correas. Lo primero es plantearse si estas correas son un objeto separado de todos los demás, que el jugador pueda coger y llevarse, o si forman parte de la propia funda, o si forman parte de la cama. Si forman parte de la cama, habrá que evitar que el jugador pueda cogerlas. Esto es fácil sin más que poner una respuesta apropiada ante la acción `Coger`, como por ejemplo: "Forman parte de la cama y no puedes llevártelas."

En cambio, si forman parte de la funda la cosa es un poco más compleja, ya que en principio el jugador no puede cogerlas, pero sin embargo si coge la funda, automáticamente debe coger también las correas. InformATE está preparado para este tipo de situaciones. Es posible indicar que un objeto tiene "componentes" inseparables de él. Así, la funda tendría como un componente las correas. La librería maneja automáticamente esta situación de modo que si el jugador intenta COGER CORREAS, le dirá "Eso forma parte de la funda." Y no le dejará. En cambio, si consigue coger la funda de alguna forma, las correas también irán con él. Y si deja

la funda en otra localidad, las correas también serán dejadas.

Para no complicar más la situación, haremos que las correas formen parte de la cama y por tanto no se puedan coger.

Una vez hayamos escrito el objeto `correas`, hay que programar el puzzle. Si el jugador pone CORTAR CORREAS CON CUCHILLO, marcaremos que las correas están cortadas (activando su atributo `general`). Esto lo haremos capturando la acción `Cortar` en el objeto `correas`. Por otra parte, el objeto `funda` debe capturar la acción `Coger` y permitir que tenga éxito sólo si las `correas` tienen activado `general` (esto es, han sido cortadas).

Ante la orden CORTAR CORREAS CON CUCHILLO, el parser generará la acción `Cortar`, con `uno=correas` y `otro=cuchillo`. Si el jugador pone CORTAR CORREAS, a secas, entonces `otro` tomará el valor cero. Y si pone CORTAR CORREAS CON CARBON, por ejemplo, entonces `otro` tomará el valor `carbon`. En nuestro código debemos comprobar que realmente `otro` es el cuchillo, o bien cero (no se ha especificado con qué cortar), y si no es ninguno de ellos imprimir un mensaje como "Eso no vale para cortar."

Si el jugador pone CORTAR CORREAS sin especificar con qué, podemos hacer que el juego responda "Tienes que especificar con qué cortarlas.", pero personalmente creo que la jugabilidad mejoraría mucho si dotamos al juego de cierta "inteligencia", de modo que él mismo asuma que quiere cortarlas con el cuchillo, si el jugador lleva el cuchillo consigo.

Así pues, las diferentes situaciones a manejar son:

```
SI ya estaban cortadas "Ya están cortadas." y terminar
SI especifica con qué cortar,
   y no es el cuchillo "Eso no corta" y terminar
SI no espeficica con qué cortar,
   y no tiene el cuchillo "Necesitas tener algo que corte" y terminar
EN OTRO CASO cortamos las correas
```

Lo que se traduce en el siguiente código Inform (fíjate en la descripción, es una rutina que cambia ligeramente según las correas estén ya cortadas o no):

```
Object correas "correas" Limbo
 with   nombre 'correas' 'correa' 'cuero',
        descripcion [;
            print "Son unas correas de cuero que ";
            if (correas has general) "cuelgan de la cama.";
            "sujetan la funda a la cama.";
        ],
        antes [;
         Coger:
            "Las correas forman parte de la cama. No puedes
            llevártelas.";
         Cortar:
            if (correas has general) "Ya las has cortado.";
            if (otro~=0 or cuchillo) "Eso no corta.";
            if ((otro==0) && (cuchillo notin jugador))
                "Necesitas tener algo cortante.";
            give correas general;
            "Cortas las correas con el cuchillo.";
        ],
 has    femenino nombreplural escenario;
```

Observa que las correas comienzan en el limbo. Deben moverse al dormitorio a la vez que la funda, es decir, al examinar el catre. De paso aprovechamos para corregir un pequeño bug que había en el catre de la versión anterior. Si te fijaste, al poner EXAMINA CATRE por primera vez, el juego dice "El catre está cubierto por una funda", y hace aparecer la funda. Al examinarlo la segunda vez dice "Sólo restos de paja cubren el catre", asumiendo que el jugador ya ha cogido la funda. Pero la descripción no es correcta si el jugador no la ha cogido. La paja debe mencionarse sólo después de que el jugador haya cogido la funda. Esto es fácil de corregir gracias a que Inform tiene un atributo llamado `movido` que se activa tan pronto como el jugador coge el objeto (en realidad este atributo podría haberse llamado `cogido` y su función sería aún más clara).

Asi que ahora el catre debe contener el siguiente código:

```
       descripcion [;
           if (catre hasnt general)
           {   ! Hacemos aparecer la funda y la correa
               move funda to Dormitorio;
               move correas to Dormitorio;
               give catre general;
           }
           if (funda has movido)
               "Sólo restos de paja cubren la cama.";
           "Una funda de tela cubre la cama.";
       ],
```

Y la funda debe ser modificada para que impida la acción `Coger` a menos que las correas hayan sido cortadas. Basta añadirle esto:

```
       antes [;
        Coger: if (correas hasnt general)
            "La funda está sujeta a la cama por unas correas.";
       ],
```

Por otro lado, si el jugador consigue coger la funda, al examinar la cama una segunda vez se le dirá que está cubierta de paja. Resulta natural que el jugador intente ahora COGER PAJA, pero ¡la paja está todavía en el Limbo! Debemos traer la paja al dormitorio en el instante en que el jugador consiga coger la funda.

Para lograrlo haremos uso de otra rutina que tienen todos los objetos y que no habíamos necesitado hasta ahora. Se trata de la rutina `despues`, que es llamada por InformATE cuando la acción ha finalizado con éxito. La rutina puede realizar las acciones que sean convenientes, imprimir un mensaje si así lo desea, y retornar TRUE o FALSE. Si retorna TRUE, la InformATE no generará ningún mensaje extra en pantalla. Si retorna FALSE, InformATE generará el mensaje de éxito habitual, por ejemplo, ante la acción `Coger` dirá "Cogido".

Así pues, podemos añadirle al objeto `funda` el siguiente código:

```
       despues [;
        Coger:
           if (paja in Limbo)
           {
               move paja to Dormitorio;
               "Con las correas cortadas logras coger la funda.";
           }
       ],
```

Debes recordar algo muy importante: La rutina `despues` solo es ejecutada para este objeto si la acción que intentó el jugador tuvo éxito. De hecho, cuando `despues` sea ejecutada, la acción ya habrá terminado, y la funda ya estará en poder del jugador. Si intentó COGER FUNDA, pero la acción fracasó, el código anterior no será ejecutado.

Fíjate que, en caso de éxito, hacemos aparecer la paja, y mostramos un mensaje. Ya que se trata de un *print implícito*, tras mostrar el texto la rutina retornará TRUE, e InformATE no añadirá nada más. Fíjate también que antes de hacer aparecer la paja y emitir el mensaje de éxito, comprobamos que la `paja` esté en el limbo. Si no lo está, no hacemos nada especial, por lo que InformATE dirá su habitual "Cogida." Este detalle es para tener en cuenta el caso de que el jugador deje la funda en otro lugar y vuelva a cogerla más tarde. En ese caso el mensaje "Con las correas cortadas logras coger la funda." ya no tendría sentido, y de hecho no aparecerá, porque en ese caso la paja estaría en el `Dormitorio` y no en el `Limbo`.

Como de costumbre, aquí tienes la versión completa del juego para que lo estudies, lo modifiques a tu antojo y lo pruebes.

La ventana y los barrotes

¡Vamos con los últimos puzzles del juego! Ahora que el jugador tiene la funda debe descubrir el barrote suelto que hay en la ventana, atar la funda a otro barrote de la misma y salir por la ventana para terminar el juego.

¿Qué objetos participan en todo esto? Evidentemente tenemos una `ventana`, un `barrote_suelto` y un `barrote_fijo`, más la `funda` que ya teníamos. Y hay una serie de acciones que debemos contemplar para cada uno de esos objetos. De modo que vayamos por partes. Lo primero será programar la ventana.

La ventana

La ventana, de acuerdo con el modelo del mundo que tiene la librería InformATE debería ser un conector que lleve de la mazmorra al exterior de la torre. Sin embargo, ya que en este juego el mero hecho de atravesar la ventana da por finalizado el juego, no necesitamos que sea realmente una conexión de esas.

De todas formas, por hacer más interesante este cursillo, crearemos una localidad nueva, el exterior de la torre, y haremos que la ventana sea un conducto que une esta localidad con la mazmorra.

Este tipo de conductos, en InformATE se llaman `puertas`, aunque como vemos no siempre son una puerta real. Son simplemente un objeto que comunica dos lugares. También podría ser un puente, una cabina de teleportación, ...

Las puertas tienen dos propiedades importantes:

- Una que indica a qué localidad conduce la puerta. Esta propiedad se llama `puerta_a`
- Otra que dice en qué lado de la habitación está la puerta, si al norte, al sur, etc.. Esta propiedad se llama `direcc_puerta`

Además, debe tener el atributo `puerta`, para indicarle a InformATE que es un conector entre dos lugares, y que por tanto debe admitir por defecto la orden ENTRAR EN o METERSE POR ese objeto.

Vamos a programar una versión inicial de la ventana, sin puzzle alguno, que deje salir al jugador de la mazmorra. En primer lugar hay que añadir a la mazmorra una nueva salida, pero en lugar de indicar a dónde lleva esa salida (llevará al exterior de la torre), ponemos qué "puerta" hay en esa salida, que en nuestro caso

sería el ventanuco. La nueva mazmorra quedará así:

```
Object  Mazmorra "Mazmorra"
 with   descripcion "Una silenciosa estancia débilmente alumbrada
            por los rayos de luna que se filtran a través de un
            pequeño ventanuco. El suelo está lleno de paja, colgando
            de unos grilletes en la pared observas un esqueleto
            humano.",
        al_e EscaleraCaracol2,
        afuera ventanuco,
 has    luz;
```

La dirección es `afuera`, y por tanto se activará cuando el jugador escriba "SALIR", o "AFUERA" (esta es una "dirección" adicional a las clásicas NORTE, SUR, etc..)

Por su parte el ventanuco debe estar localizado en la `Mazmorra`, y su código sería el siguiente:

```
Object ventanuco "ventanuco" Mazmorra
 with   nombre 'ventanuco' 'ventana' 'tragaluz',
        descripcion "A través de los barrotes de este ventanuco
            puedes ver el exterior de la torre, iluminado por una
            increíble luna llena.",
        puerta_a ExteriorTorre,
        direcc_puerta afuera,
 has    puerta abierta escenario;
```

Fijate que la propiedad `puerta_a` indica que esta ventana conduce a la localidad `ExteriorTorre` cuando se la atraviese. De modo que si el jugador está en la Mazmorra y pone "SALIR", esto es lo que ocurrirá:

- El parser generará la acción `Ir` con `uno=obj_afuera` (`obj_afuera` es una forma que tiene Inform de representar la "dirección exterior", igual que tiene `obj_n` para representar la "dirección norte", etc).
- Para llevar a cabo esa acción, InformATE mirará en primer lugar a dónde conduce la salida `afuera` de la mazmorra, y encontrará que conduce al ventanuco.
- Seguidamente verá que el ventanuco tiene el atributo `puerta`, de donde deducirá que no se trata del destino del viaje, sino de un mero tránsito. Ya que tiene el atributo `abierta`, el jugador puede pasar a través de ella.
- Entonces InformATE buscará en la propiedad `puerta_a` del ventanuco a dónde debe ir a parar. Encontrará allí `ExteriorTorre` que ya es una localidad normal, de modo que moverá al jugador allí.

Si queremos impedir que el jugador salga antes de haber reunido ciertas condiciones, podremos escribir una rutina en la salida `afuera` de la localidad. Esta rutina puede imprimir un mensaje y retornar true para impedir al jugador que salga, o bien retornar `ventanuco` para que todo prosiga normalmente. Por ejemplo, el código siguiente no dejará salir al jugador mientras no lleve consigo el carbón (es un ejemplo tonto, un ejemplo más realista lo programaremos más adelante, donde comprobaremos si el jugador ha aflojado el barrote y ha atado la funda antes de dejarle salir).

```
        afuera [;
            if (carbon in jugador) return ventanuco;
            else "Tienes la sensación de que olvidas algo en esta
                  torre. Decides no irte todavía.";
        ],
```

Pero observa de nuevo el código del ventanuco. Además, de indicar a qué localidad lleva, es necesario poner en el ventanuco la propiedad `direcc_puerta`, con el valor `afuera`. El por qué es necesaria esta propiedad es un poco rebuscado, y en mi opinión es un error de diseño de InformATE, pero ya que las cosas están así, mejor que estés al tanto de esto. La razón es que el jugador puede, en lugar de poner SALIR, poner algo como METERSE POR LA VENTANA. Esto genera una acción `Meterse`, con `uno=ventanuco`, en lugar de generar `Ir` con `uno=obj_afuera`. Si InformATE se limitara a mirar la dirección `puerta_a`, y llevar al jugador al `ExteriorTorre`, entonces la rutina que hemos programado para impedir que el jugador pueda salir, no sería ejecutada. Dicho de otro modo, sería diferente lo que ocurre cuando el jugador pone SALIR que cuando pone METERSE POR EL VENTANUCO. Esto no estaría bien.

Por mantener una cierta coherencia, InformATE quiere asegurar que siempre que el jugador se mueva de un lugar a otro, se generará una acción `Ir`, tanto si pone SALIR, como si pone METERSE POR EL VENTANUCO. Para ello, lo que hace InformATE cuando el jugador intenta meterse por una puerta, es consultar la propiedad `direcc_puerta` de dicha puerta, y generar una acción `Ir` que lleve a esa dirección. Así que en el caso de nuestro ventanuco, si el astuto jugador pone METERSE POR VENTANUCO, InformATE cambiará esa acción en IR AFUERA, por lo que la rutina `afuera` que hemos visto antes se ejecturará y podremos impedir al jugador que salga si no cumple las condiciones que le hemos impuesto.

Resumen para programar una conexión:

- En la localidad que tiene esa conexión, una de sus salidas debe llevar a la conexión en cuestión. Por ejemplo, la salida `al_n` puede llevar a un `Porton`.
- Hay que programar un objeto conexión (el `Porton`) que tenga las propiedades:
  - `puerta_a` que indica a qué localidad destino lleva la puerta
  - `direcc_puerta` que indica en qué punto cardinal de la localidad origen se halla la puerta (`al_n`)

  y los atributos:
  - `puerta` para que InformATE sepa que es una conexión, y no una localidad.
  - `abierta` para que el jugador pueda pasar
  - o bien `abrible` para que el jugador pueda abrirla o cerrarla (puede hacerse también que requiera una llave, pero no lo compliquemos más).
  - `escenario` ¡para que el jugador no pueda llevarse la conexión debajo del brazo!

Las conexiones (o puertas) son ciertamente un objeto complicado en InformATE. Ten en cuenta que todo lo anterior es necesario para hacer una puerta que lleve de A a B, pero ¡no al revés! Si quieres volver de B a A, ¡necesitas programar otra puerta en sentido contrario! Puede hacerse un solo objeto que lleve en ambas direcciones a la vez, pero esto exige una programación aún más retorcida. Por todo esto se ha desarrollado un módulo de ampliación, llamado "Puertas", que intenta simplificar al máximo la programación de puertas con dos lados. Si estás interesado, puedes descargar este módulo de la zona de modulos de InformATE!.

Los barrotes

Inicialmente, sólo estará presente el objeto "barrotes". Cuando el jugador lo examine, haremos aparecer el objeto "barrote flojo", para que el jugador pueda retirarlo. Esto planteará un problema de ambigüedad si el jugador intenta despues ROMPER BARROTE o ATAR FUNDA A BARROTE. ¿A cuál se refiere? ¿Al flojo que tiene en la mano, o a uno de los barrotes que han quedado en la ventana?

Por suerte InformATE maneja las ambigüedades de forma automática. Sólo tenemos que tener la precaución de dar un nombre corto diferente a cada uno de los objetos que podrían dar lugar a confusión, y que su lista de sinónimos no sea idéntica. Así, por ejemplo, el barrote flojo puede tener como nombre "barra", y en su lista de sinónimos pondremos la palabra 'flojo' y 'floja', mientras que los restantes barrotes los llamaremos "barrotes sólidos" y pondremos las palabras 'solido' y 'fijo' en su lista de sinónimos.

Vamos a programar los barrotes sólidos, para que al ser examinados hagan aparecer el barrote flojo. Se trata del mismo truco que ya usamos para hacer aparecer el cuchillo al examinar el esqueleto, es decir, hacer uso del atributo `general`. Además, usamos el atributo `movido` del barrote flojo, para saber si el jugador lo ha retirado ya o no, y así cambiar la descripción del ventanuco.

```
Object barrotes "barrotes sólidos" Mazmorra
 with   nombre 'barrotes' 'barrote' 'solido' 'barra' 'barras'
            'solidos' 'fijos' 'fijo',
        descripcion [;
            if (barrotes hasnt general)
            {
                move barrote_flojo to Mazmorra;
                give barrotes general;
                "Al examinar de cerca los barrotes de la ventana,
                descubres uno que parece estar más flojo.";
            }
            if (barrote_flojo has movido)
                "En la ventana falta un barrote. Parece que podrías
                pasar por el hueco.";
            "Entre los barrotes hay uno que parece más flojo.";
        ],
 has    nombreplural escenario;
```

Fijate que le hemos puesto el atributo `escenario`, para impedir que el jugador pueda coger estos barrotes. En cuanto al `barrote_flojo`, inicalmente estará en el limbo. Usaremos de nuevo su atributo `movido` para saber si el jugador lo ha retirado ya de la ventana o no, y así cambiar su descripción. En este caso no le damos el atributo `escenario`, para que el jugador pueda cogerlo. Pero le damos el atributo `oculto`, para que al MIRAR, la descripción de la localidad no diga "Puedes ver un barrote flojo". Fijate que hemos capturado algunas acciones obvias que el jugador intentará con el barrote, como moverlo, empujarlo... Y hemos cambiado tambien el mensaje por defecto para cuando el jugador lo coja.

```
Object barrote_flojo "barra" Limbo
 with   nombre 'barrote' 'flojo' 'barra' 'floja',
        descripcion [;
            if (barrote_flojo has movido)
                "Es el barrote que quitaste del ventanuco de la
                mazmorra.";
            "Parece que este barrote podría quitarse con un poco de
            esfuerzo.";
        ],
        antes [;
         Empujar, Tirar, Mover: "Aflojas el barrote.";
```

```
       ],
       despues [;
        Coger: "Con esfuerzo, consigues arrancar el barrote de su
           sitio.";
       ],
has    femenino oculto;
```

Fíjate que le hemos puesto como nombre corto "barra", y eso nos obliga a ponerle el atributo `femenino`. De ese modo, cuando el jugador lo coja y mire su inventario, InformATE le dirá que lleva "una barra", y no "un barra".

Atando cabos

Sólo queda por resolver la acción de ATAR FUNDA A BARROTE. InformATE tiene ya programado el verbo ATA, y genera la acción `ATAR`. Al igual que ocurre con `Cortar`, el jugador puede poner simplemente ATA FUNDA, o bien ATA FUNDA A BARROTE. En el primer caso la variable `otro` tomará el valor cero, y en el segundo tomará el valor `barrote_flojo`. Si pone ATA FUNDA A BARROTES, `otro` tomará el valor `barrotes`.

En todos los casos, el objeto `uno` es la `funda`, y es quien recibirá la acción. Por tanto la respuesta a `Atar` debemos programarla en la funda, y no en los barrotes. Por tanto debemos modificar la rutina `antes` de la funda, para tratar en ella las diferentes ataduras que se le pueden ocurrir al jugador, y dar un mensaje adecuado en cada caso. Cuando el `otro` sean los barrotes, marcaremos el puzzle de la funda como resuelta, activando la propiedad `general` de la `funda`. Observa que no podemos usar la propiedad `general` de los barrotes porque esa ya la hemos usado para marcar si han sido examinados o no.

La nueva rutina `antes` de la funda debería cambiar también la respuesta ante `Coger`, comprobando si la funda está atada a los barrotes, ya que en ese caso el jugador no podrá cogerla. También tenemos que manejar correctamente la acción `Desatar`. La rutina `antes` que se ocupa de todo esto podría ser la siguiente:

```
       antes [;
        Coger:
           if (correas hasnt general)
               "La funda está sujeta a la cama por unas correas.";
           if (funda has general)
               "La funda está atada a los barrotes.";
        Atar:
           if (otro==0)
               "No tiene sentido hacer un nudo en la funda.";
           if (otro==correas or catre or esqueleto or cuchillo)
               "No tiene sentido atar la funda ahí.";
           if (otro==carbon or antorcha)
               "¿Estás loco?";
           if (otro==barrote_flojo)
               "Mejor atarla a un barrote sólido.";
           if (otro==barrotes or ventanuco)
           {
               give funda general;
               move funda to localizacion;
               "Atas la funda a uno de los barrotes más sólidos y
               la dejas colgando por la ventana.";
           }
       ],
```

Fijate en un detalle. Cuando el jugador ATA con exito la funda a los barrotes, no sólo le damos el atributo

`general` para marcar este hecho. Además, movemos la funda a la localización actual (que será la mazmorra, ya que de lo contrario los barrotes no estarían presentes y la acción no se habría generado). Esto lo hacemos para evitar que la funda siga en poder del jugador despues de haberla atado. Si no lo hicieramos, el jugador podría salir de la mazmorra y llevarse la funda consigo ¡mientras aún está atada a los barrotes!

Ya solo queda modificar la salida `afuera` de la mazmorra para que sólo deje salir al jugador cuando este haya resuelto el puzzle de los barrotes. Vamos allá:

```
       afuera [;
           if (barrote_flojo hasnt movido)
               "No puedes salir por el ventanuco, los barrotes no
               dejan suficiente hueco.";
           if (funda hasnt general)
               "No te atreves a saltar desde esta altura. Necesitas
               algún tipo de cuerda para descolgarte.";
           print "Contorsionándote, logras hacer pasar tu cuerpo por
               la estrecha abertura que dejó el barrote, y te
               descuelgas con cuidado usando la funda a modo de
               cuerda.^^";
           return ventanuco;
```

El final del juego

Bueno, si el jugador hace todo bien, conseguirá llegar a la localidad "Exterior de la torre", y se supone que ahí termina el juego. Pero si juegas lo anterior verás que el juego sigue esperando nuevos comandos del jugador, aunque realmente ya nada puede hacerse en esa localidad sin salidas.

La forma de dar por terminado el juego es hacer la asignación siguiente:

```
banderafin=2;
```

Si al final del turno, InformATE encuentra que `banderafin` vale 2, entiende que el juego ha terminado con éxito. Si vale 1, entiende que ha terminado con la muerte del aventurero. En ambos casos ya no le pide más órdenes, sino que le invita a rejugar la aventura o finalizar el programa, con un mensaje apropiado a cada caso.

El lugar obvio para hacer esa asignación es el momento en que el jugador sale por la ventana, esto es, en la rutina `afuera` de la mazmorra, cuando ha impreso el mensaje que explica cómo sale el jugador por la ventana, y va a retornar el objeto `ventanuco`.

La nueva rutina queda entonces así:

```
       afuera [;
           if (barrote_flojo hasnt movido)
               "No puedes salir por el ventanuco, los barrotes no
               dejan suficiente hueco.";
           if (funda hasnt general)
               "No te atreves a saltar desde esta altura. Necesitas
               algún tipo de cuerda para descolgarte.";
           print "Contorsionándote, logras hacer pasar tu cuerpo por
               la estrecha abertura que dejó el barrote, y te
               descuelgas con cuidado usando la funda a modo de
               cuerda.^^";
           banderafin=2;
           return ventanuco;
```

Parte III − Puliendo detalles

Aunque el juego ya está completo, en el sentido de que puede ser finalizado por quien conozca la solución, ¡no está aún terminado ni mucho menos! Hay cantidad de detalles sin pulir. Para empezar, hay cantidad de cosas que se mencionan en las descripciones de los lugares, y que después no están programadas, por lo que el jugador no podrá ni siquiera examinarlas (por ejemplo, la paja en la mazmorra, la mesa de guardia, la armadura en la escalera,...) El siguiente paso sería programar esos objetos para que al menos tengan una descripción. Lo más sencillo es ponerlos con el atributo `escenario`, y así la InformATE no dejará que el jugador los manipule, y dará un mensaje apropiado ante cada posible acción. De todas formas, a lo mejor quieres cambiar alguno de los mensajes por defecto para algunas de esas acciones, para dar mensajes más adecuados. ¡Ya sabes cómo! Basta escribir una rutina `antes` en el objeto que capture esa acción.

Por otro lado, los jugadores pueden tener una imaginación muy calenturienta a la hora de probar soluciones raras. A mí se me ocurren ahora mismo un montón de cosas que el jugador podría hacer y que no hemos contemplado. Por ejemplo, quemar la funda con la antorcha, cortar la funda con el cuchillo, desatar la funda de la ventana una vez que la ha atado... Deberías intentar esas cosas y ver qué responde el juego. Si la respuesta no es muy apropiada ¡cambiala!

También es posible que al jugador no se le ocurran exactamente las palabras que nosotros hemos pensado para resolver los puzzles. Por ejemplo, una vez que ha atado la funda a los barrotes, si en lugar de SALIR pone BAJAR ¿funcionará? ¿Y si pone BAJAR POR LA FUNDA? Y para atar la funda hemos previsto ATAR FUNDA A BARROTES, pero ¿y si pone ATAR FUNDA EN BARROTES? En este punto suele ser muy útil el comando de depuración ACCIONES que ya hemos visto, para detectar qué acciones se generan cuando se ponen diferentes frases, y así poder tratar también esos casos.

Hay verbos que no están contemplados en InformATE. Por ejemplo, y curiosamente, no está contemplado el verbo DESATAR (aunque sí ATAR). De modo que si el jugador intenta DESATAR LA FUNDA, le dirá que no entiende ese verbo. Corregir este punto implica definir un nuevo verbo y una nueva acción. Esto no es nada difícil, y ahora veremos cómo se hace.

Definiendo nuevos verbos y extendiendo los existentes

Hemos mencionado en la sección anterior unos cuantos verbos que se le pueden ocurrir al jugador. Uno de ellos es, por ejemplo DESATAR FUNDA. La respuesta por defecto de InformATE ante ese comando es "No conozco ese verbo."

La librería InformATE proporciona una buena cantidad de verbos ya progrmados, pero tarde o temprano llegarás a uno que no estaba programado. Aquí mostraremos cómo añadir un verbo nuevo al lenguaje manejado por InformATE.

Verbo nuevo, acción vieja

La primera cuestión que debes plantearte es: además de un nuevo verbo, ¿necesito una nueva acción? Hay casos en los que no es así, sino que un verbo nuevo da lugar a una de las acciones previamente existentes. Este es el caso más fácil de manejar.

Por ejemplo, el verbo MORDER no está previsto en InformATE. Si el jugador intenta MORDER BARROTES, le dirá que no comprende ese verbo. Podemos añadir ese verbo para que genere la acción Comer, que ya existe en InformATE, asociada al verbo COMER. Para lograr esto, la cosa es tan simple como poner lo siguiente después de `Include "Gramatica"`

```
Verb 'muerde' = 'come';
```

Un detalle importantísimo: siempre que añadas un nuevo verbo, debes hacerlo en IMPERATIVO, y no en infinitivo. No debes poner 'morder' sino 'muerde'. Y la hora de hacerlo sinónimo de un verbo preexistente, el verbo original también estará en imperativo, es decir 'come' en lugar de 'comer'.

Ahora el juego ya comprenderá MUERDE BARROTES, generará la acción `Comer barrotes`, y producirá la respuesta por defecto "Eso es simplemente incomestible". Si quieres otra respuesta basta ponerla en la rutina `antes` de los barrotes, para la acción `Comer`.

Sin embargo el juego no comprende MORDER BARROTES. El verbo que hemos añadido está en forma imperativa. InformATE es capaz de deducir por sí solo la forma infinitiva si el verbo es regular, ya que en este caso basta añadir una R. Por ejemplo, "come" se convierte en "comer". En cambio "muerde" se convertiría en "muerder", por lo que InformATE no sabe hacerlo sin tu ayuda. Esto no sólo afecta a los comandos que escribe el jugador, sino a algunas frases que InformATE genera por sí solo a veces. Por ejemplo, si el jugador pone MUERDE sin más, InformATE detectará que la frase está incompleta, y le preguntará "¿Qué quieres muerder?", de nuevo en este caso genera la forma infinitiva símplemente añadiendo una R a la forma imperativa, que es la que el programador ha definido.

Para evitar ambos errores, el programador debe decirle a InformATE que `morder` es un verbo irregular. Para esto, basta añadir la siguiente línea a continuación:

```
VerboIrregular 'morder' with imperativo 'muerde';
```

Recuerda siempre que definas un nuevo verbo, comprobar si es regular, es decir, si al añadirle una R sale el infinitivo. Si no lo es, debes definir su forma infinitiva mediante una línea como la anterior. Si no lo haces el juego no comprenderá el comando en inifinitivo.

De este error nadie está libre. El propio autor de este cursillo olvidó declarar como irregular el verbo `regar` en la Aventura original, y como consecuencia el juego sólo admitía la forma imperativa "riega". (Bueno, y también admitía la forma aberrante "riegar", pues sale añadiendo una R a lo que InformATE conocía)

Verbo nuevo, acción nueva

Un caso más complejo aparece con DESATAR. Este es un verbo nuevo, pero la acción que debe generar es también nueva. No hay acción `Desatar` en InformATE, y no parece apropiado convertir la orden DESATA FUNDA en otra cosa rara como "mover funda". Lo propio será definir la nueva acción y asociar el nuevo

verbo con ella.

Definir una acción nueva consiste en programar una rutina cuyo nombre ha de ser el mismo que la acción, más la palabra `Sub`. En nuestro ejemplo el nombre de la rutina sería `DesatarSub`. Esta rutina será ejecutada como "acción por defecto". Es decir, si el jugador pone DESATAR FUNDA, el parser generará la acción `Desatar` con `uno=funda` y como ya sabemos esto da lugar a una serie de "avisos" que el parser envía a los objetos cercanos a través de la rutina `reaccionar_antes` de estos objetos. Si ninguno de los objetos "dice nada", entonces finalmente se llamará a la rutina `antes` del propio objeto que el jugador ha nombrado (la funda). Si esta rutina tampoco "dice nada" (es decir, si no retorna `true`), será entonces cuando InformATE ejecutará la rutina `DesatarSub` y dará el asunto por terminado.

Por tanto, la rutina `DesatarSub` hay que programarla bajo este punto de vista: se ejecutará cuando el jugador intente "desatar algo", y ese "algo" no maneja por sí mismo la situación. Hay dos alternativas a la hora de programar una rutina de acción:

- La más simple es hacer que la rutina simplemente emita un mensaje. Por ejemplo: "No sé como desatar eso." y termine. En este caso, ese será el mensaje que se verá cada vez que el jugador intente DESATAR ALGO, y si queremos cambiar ese mensaje, bastará capturar la acción `Desatar` desde la rutina `antes` del objeto en cuestión.
- La más complicada es que la rutina haga algo realmente. Por ejemplo, la rutina asociada a la acción `Encender` sí que hace algo (mira si el objeto en cuestión es `conmutable`, si está ya `encendido`, mostrando mensajes apropiados en cada caso, o bien cambiando su atributo `encendido` si procede).

En InformATE, aunque la acción `Atar` existe, en realidad la libería no prevé que haya objetos que puedan ser atados, en la misma forma que prevé que hay objetos que pueden ser abiertos/cerrados o encendidos/apagados. La rutina `AtarSub` que ya viene con InformATE es "de las que no hacen nada", sino que simplemente emite el mensaje "No lograrás nada así" cada vez que el jugador intente atar algo.

Si nuestro juego va a contener gran cantidad de cuerdas, que pueden ser atadas y desatadas a diferentes objetos del juego, habría que plantearse añadir este nuevo tipo de objeto al modelo del mundo. Habría que "inventar" el tipo de objetos "atables", y entonces cuando el jugador ponga ATAR `uno` A `otro`, la rutina `AtarSub` debería comprobar si `uno` es del tipo `atable`, y si lo es, modificar de alguna forma su estado para señalar que ha quedado atado al `otro`. Esto puede tener implicaciones profundas, por ejemplo ¿qué pasaría cuando el jugador intenta moverse de una localidad, llevando en su mano una cuerda que está atada a un objeto fijo de esa localidad? ¿Se le debe permitir? ¿Habrá que tener en cuenta la longitud de la cuerda? ¿Y si una cuerda se ata a otra cuerda?

El manejo de cuerdas u objetos atables de forma totalmente genérica, es un asunto bastante difícil que se sale de los objetivos de este cursillo. Por suerte, en nuestro juego el único objeto que puede ser atado es la funda, y además no necesitamos que pueda atarse a cualquier cosa, sino solo que pueda atarse al barrote. Así que ese caso particular podemos manejarlo, como hemos hecho en el capítulo anterior, desde la rutina `antes` de la propia funda.

Por tanto, nuestra rutina `DesatarSub` será también de las que "no hacen nada" (o sea, simplemente emiten un mensaje). Y dejaremos que la propia funda maneje la situación desde su rutina `antes`. Así que lo único a decidir es qué mensaje poner cuando el jugador intente DESATAR ALGO que no sea la funda. Digamos que vamos a poner "Eso no puede desatarse". Entonces la rutina será la siguiente:

```
[ DesatarSub ;
   print_ret "Eso no puede desatarse.";
];
```

Por el mero hecho de haber programado una rutina llamada `DesatarSub`, tenemos ya una nueva acción llamada `Desatar`. Esa acción puede manejarse desde la rutina `antes` de cualquier objeto, y en particular, desde la funda. Para poner algo como esto:

```
        Desatar:
           if (funda hasnt general)
               "La funda no está atada a ningún sitio.";
           give funda ~general;
           "Desatas la funda de los barrotes.";
```

Observa que este código está escrito específicamente para este juego, ya que si la funda no tiene el atributo `general`, es que no ha sido atada a nada, y si lo tiene es que está atada a los barrotes, ya que en este juego no dejamos que el jugador ate la funda a ningún otro lugar. Si quisiéramos hacer un juego más flexible, deberíamos dejar que el jugador pudiera atar la funda a donde quisiera, aunque no sirviera para nada. Por ejemplo, puede ocurrirsele atar la funda al esqueleto. Esto complicaría las cosas, ya que no basta con tener un atributo como `general` para saber si la funda está atada o no, sino que además habría que tener una nueva propiedad que almacene a qué objeto está atada. Cuanta más flexibilidad se le quiera dar al jugador, más se complica la programación.

Ya solo queda una cosa por hacer. La más importante de todas. Se trata de asociar un verbo con esta nueva acción. Si olvidamos hacerla, el jugador no podrá nunca DESATAR LA FUNDA, ya que, aunque el juego prevé la acción `desatar`, esa acción no se genera nunca, ya que no hay ningún verbo que la genere. Nada de lo que pueda escribir el jugador daría lugar a esa acción.

Definir una asociación entre verbo y acción es lo que en InformATE se conoce como "definir una gramática". En realidad, la gramática no asocia un verbo con una acción, sino un "tipo de frase" con una acción. Por ejemplo, en nuestro caso queremos que entienda frases como DESATA FUNDA, y DESATA FUNDA DE BARROTE (tal como hemos programado la acción `Desatar` en la funda, el jugador no necesita especificar el barrote, pues lo damos por supuesto, pero si de todas formas decide ponerlo, el juego debería aceptar la orden).

Por tanto tenemos dos "tipos de frase" que el juego debe comprender. Ambos tipos comienzan por el verbo "desata", y después:

- El primero requiere un nombre de un objeto (p.ej: "funda")
- El segundo requiere un nombre de un objeto, la preposición "de", y un nombre de otro objeto.

InformATE permite asociar a cada tipo de frase una acción diferente, pero en este caso particular no nos interesa. Ambos tipos de frase generan la misma acción `Desatar`. Así que la "gramática" que tenemos que escribir en nuestro juego (después del `Include "Gramatica"`) será:

```
Verb 'desata'
  *  noun                -> Desatar
  *  noun 'de' noun      -> Desatar;
```

Analicemos qué significa esto.

- En primer lugar aparece la palabra `Verb` que indica que vamos a definir un nuevo verbo.
- Seguidamente `'desata'` entre comillas simples, es el verbo a definir. Recuerda que debes ponerlo siempre en forma imperativa (esto es, no 'desatar'). Esta es la palabra que tendrá que teclear el jugador. Podemos poner varias, separadas por espacios, como en la propiedad `nombre` de un objeto. En ese caso se entiende que todas esas palabras son el mismo verbo.

- A continuación van una serie de líneas, cada una de las cuales comienza por un asterisco, y la última de ellas termina por punto y coma. (Ojo, las anteriores a la última **no terminan por punto y coma**).
- La primera de estas líneas simplemente dice `noun`, una flecha, y `Desatar`. Lo que significa es que `noun` puede ser el nombre de cualquier objeto que el jugador tenga a su alrededor. Es decir, podría ser algo tan simple como "funda", o tan complejo como "espada magica con incrustaciones de rubi". El parser decidirá que una serie de palabras más o menos larga es un nombre de objeto, si todas esas palabras aparecen en la propiedad `nombre` de un objeto.

  Si el parser llega al final de la frase y todas las palabras tecleadas por el jugador le "encajan" como nombre de objeto, el parsing ha tenido éxito, y la acción generada será `Desatar` (la que se indique después de la flecha), y la variable `uno` apuntará al objeto cuyo nombre ha encajado con todo eso.

- La siguiente línea es parecida, sólo que `noun` aparece dos veces, con la palabra `'de'` en medio (ojo, esta palabra debe ir con comillas simples). Así que el parser espera ahora una secuencia de palabras que se refieran a un objeto, seguida de la palabra 'de', seguida de otra secuencia de palabras que se refieran a otro objeto (o tal vez al mismo). Por ejemplo: DESATA FUNDA SUCIA DE BARROTE FIJO. En este caso, si el objeto `funda` tiene en su nombre las palabras 'funda' y 'sucia', las palabras "funda sucia" serán comprendidas como referidas a ese objeto, y la variable `uno` apuntará a él. Después aparece la preposición 'de', tal como el parser espera. Finalmente aparecen las palabras "barrote fijo", que el parser intentará encajar con el `nombre` de algún objeto cercano. Si el barrote está cerca, finalmente se generará la acción `Desatar` con `uno==funda`, `otro==barrote_fijo`.

Observa que si el jugador pone DESATA FUNDA DE BARROTE, pero el barrote no está en la misma habitación que el jugador, el parser no podrá entender la palabra "barrote", pues no pertenece a ningún objeto cercano. En este caso el parser da el error "No veo eso por aquí", **y la acción no se genera**.

En Inform, las acciones sólo se generan si el parser tiene éxito al tratar de comprender la frase. Y el parser sólo tendrá éxito si todos los objetos que se mencionan en la frase están disponibles para el jugador.

Esto ahorra al programador el tener que comprobar si el jugador está en la misma localidad que la funda, y cosas así que eran típicas en PAWS.

El parser de InformATE es inteligente con respecto a la preposición "de". Imagina que en el juego tienes una "funda de plástico". ¿Cómo sabe el parser si DESATA FUNDA DE PLASTICO pertenece a una frase del tipo 1 (desata cosa) o del tipo 2 (desata cosa de cosa)?

¿Y si ponemos DESATA FUNDA DE PLASTICO DE BARROTE DE HIERRO? ¡Qué locura! ¡Cuántas veces aparece la preposición "de"! No te preocupes, el parser no se confundirá, e interpretará "funda de plastico" como el primer nombre y "barrote de hierro" como el segundo, con tal de que el objeto `funda` tenga en su propiedad `nombre` las palabras 'funda' y 'plastico', y el objeto `barrote_fijo` tenga las palabras 'barrote' y 'hierro'.

El truco que usa InformATE para no liarse con los "de", es que simplemente los ignora, si las palabras que tiene alrededor del "de" son nombres de un mismo objeto. Así, en DESATA FUNDA DE PLASTICO DE BARROTE DE HIERRO, el primer DE es ignorado, pues las palabras FUNDA y PLASTICO son del mismo objeto. El segundo DE no es ignorado, pues las palabras PLASTICO y BARROTE no son del mismo objeto. Finalmente, el tercer DE sí es ignorado, porque BARROTE y HIERRO son del mismo objeto. Así que la frase final queda

como DESATA FUNDA PLASTICO DE BARROTE HIERRO.

Redefinir o extender verbos

¿Y si tanto el verbo como la acción ya estaban definidos en InformATE, pero no hacen lo que yo quiero que hagan, o les falta algún "tipo de frase"?

Por ejemplo, el verbo 'ata' ya está definido, y la acción `Atar` también. Sin embargo el juego no entiende ATA FUNDA EN BARROTE. ¿Por qué? ¿Qué hacer?

La razón por la que no lo entiende, es porque InformATE sólo define algunos "tipos de frase" para el verbo 'ata', que son los siguientes (el siguiente fragmento está sacado del fichero `Gramatica.h`):

```
Verb 'ata' 'enlaza' 'enchufa' 'une'
    * noun                              -> Atar
    * 'a//' creature                    -> Atar
    * 'a//' creature 'a//' noun         -> Atar
    * noun 'a//' noun                   -> Atar;
```

Vemos aquí que InformATE define como sinónimo de 'ata', el verbo 'enchufa' y 'une'. Esto significa que, curiosamente, en nuestro juego de La Torre, el jugador puede escribir ENCHUFA FUNDA A BARROTE, ¡y funcionará!

Pero olvidémonos de momento de eso, y veamos el resto de la gramática del verbo 'ata'. Observa que se definen para este verbo cuatro "tipos de frase", y todas ellas generan la misma acción `Atar`. La primera es un simple `noun`, para admitir cosas como ATA FUNDA.

La segunda tiene un `'a//'`. La doble barra es una rareza de Inform. Necesitas ponerla si la preposición que quieres especificar consta de una sola letra. Si te olvidas de esa doble barra, en el fondo esta línea significa que tras 'ata', debe venir la palabra 'a', y despues una criatura. `creature` es como `noun`, solo que además el parser sólo aceptará la orden si el objeto que le sale tiene el atributo `animado` (no hemos visto nada de este atributo, pero es básico para introducir otros personajes en la aventura). Por tanto esta línea está pensada para admitir comandos como ATA A THORIN. En cambio rechazará la orden ATA A FUNDA, ya que FUNDA no es un ser animado (el parser dirá "Sólo puedes hacer eso con seres animados").

La tercera es similar a la anterior, sólo que permite especificar un segundo objeto. Por ejemplo ATA A THORIN A LA ESTACA.

Por cierto, no debes preocuparte si el jugador usa 'AL' en lugar de 'A', en una frase como ATA AL ORCO AL PALO. InformATE convierte esos 'AL' en 'A' antes de ponerse a interpretar la frase. Lo mismo cabe decir para los 'DEL', que son convertidos en 'DE'.

Finalmente, la cuarta es para frases del tipo ATA FUNDA A BARROTE.

Como vemos, no se especifica ninguna frase del tipo ATA FUNDA EN BARROTE, y por eso esta orden fracasa.

Por suerte es fácil "extender" un verbo para añadirle más "tipos de frase". Una forma sería abrir el fichero `Gramatica.h`, buscar las líneas donde está definido el verbo 'ata', y añadirle tú mismo nuevas líneas. Esta forma se desaconseja, no conviene tocar los ficheros que forman parte de `InformATE`, porque si quieres compartir tu código fuente con otras personas y has modificado la librería, deberías indicar a esas otras personas qué modificaciones has hecho para que ellos las hagan también. A la larga sería un caos.

InformATE proporciona un método para extender gramáticas sin necesidad de retocar el fichero `Gramatica.h`. Consiste en hacer uso de la palabra especial `Extend`, y seguidamente definir las nuevas gramáticas usando la misma sintaxis que para definir un verbo nuevo.

Así que en nuestro caso, bastaría añadir lo siguiente tras el `Include "Gramatica.h"`:

```
Extend 'ata'
  * noun 'en' noun          -> Atar;
```

Esto añade un quinto "tipo de frase" a los ya definidos. Observa que si quisiéramos ser coherentes con lo que ya está definido en InformATE, no estaría de más añadir un sexto tipo, para seres animados, que admita el comando ATA A THORIN EN LA ESTACA. ¿Cómo se haría?

El usar la palabra especial `Extend`, tiene el mismo efecto que añadir la línea en cuestión en el fichero `Gramatica.h`, al final de las que ya había. Si en lugar de añadirla al final, quisiéramos añadirla al principio, de modo que nuestra línea sea el "primer tipo de frase" para ese verbo, habría que poner

```
Extend 'ata' first
  * noun 'en' noun          -> Atar;
```

La diferencia entre ponerla la primera o la última no es importante en este caso, pero en otros sí tiene su importancia. Debes saber que el parser de InformATE va probando los diferentes "tipos de frase" en el orden en que los encuentra, y tan pronto como uno de ellos "encaje perfectamente", ya no seguirá mirando los demás.

Detalles avanzados sobre gramáticas

En la gramática que hemos visto, la palabra especial `noun` representa al nombre de cualquier objeto al que el jugador pueda referirse. InformATE sólo aceptará la orden del jugador si es capaz de encontrar un objeto "en las inmediaciones" cuyo nombre coincida con lo tecleado por el jugador, que puede ser una sola palabra como "COFRE" o varias como "COFRE DE MADERA DE ROBLE".

En el párrafo anterior hay un concepto por explicar, y es ¿Qué son exactamente "las inmediaciones" del jugador? Dicho de otro modo, si el jugador lleva una bolsa y dentro de ella hay un trozo de carbón, ¿entenderá el parser "EXAMINA TROZO DE CARBON"? ¿Puede considerarse que el trozo de carbón aún está "en las inmediaciones del jugador" incluso si está encerrado en una bolsa donde no puede verse?

La respuesta a la pregunta anterior es complicada. InformATE considera por defecto que "las inmediaciones del jugador" se componen de todos los objetos que hay en la misma localidad que el jugador, más todos los objetos que hay en el inventario del jugador, más todos los objetos que pueda haber dentro de cualquiera de estos objetos, a menos que se hallen en un recipiente cerrado que no sea transparente. Así que la orden EXAMINA TROZO DE CARBON será comprendida por el parser sólo si el trozo de carbón está en la misma localidad, o si el jugador lo lleva consigo, o si está dentro de un recipiente abierto o dentro de un recipiente cerrado transparente. Si no se cumple lo anterior, es decir, si el carbón por ejemplo está en una bolsa cerrada opaca, el parser directamente **¡no comprenderá la frase!**

Si el objeto al que se refiere el jugador no está en las inmediaciones, no es que la acción fracase. ¡Es que ni siquiera se produce ninguna acción, ya que el juego simplemente no entiende lo que el jugador quiere decirle!

Por otro lado, quizás en algún juego necesites acciones "raras" que puedan operar sobre objetos que no están en las inmediaciones. Por ejemplo, imagina un juego que tenga el verbo "TELEPORTA", que pueda operar

sobre un objeto que está en otra localidad, de modo que si el jugador pone TELEPORTA CARBON, el carbón aparezca en su mano. Un verbo así es mucho más difícil de programar. Podemos intentar el enfoque que hemos aprendido antes y definir el verbo y su acción asociada de esta forma:

```
Verb 'teleporta'
  *  noun            -> Teleportar;

[ TeleportarSub;
  move uno to jugador;
  "Aplicando tu poder de teleportación, te concentras y a los
   pocos segundos ", (el) uno, " se materializa en tu mano.";
];
```

Pero sin embargo no funcionará como queremos. Si el jugador pone TELEPORTA CARBON cuando el carbón no está "en las inmediaciones", el parser directamente no entenderá la frase, y la acción no se producirá. Por tanto no llegará a ejecutarse nunca la rutina `TeleportarSub`. El jugador sólo podrá usar este verbo sobre objetos que estén en sus inmediaciones ¡pero eso le quita toda la gracia!

InformATE tiene previsto que algunos juegos puedan necesitar cosas así, de modo que permite al programador que defina como quiera "qué son las inmediaciones". Así, para un verbo concreto (por ejemplo para el verbo "teleporta"), las inmediaciones puedan ser el mundo entero y todos los objetos que contiene, mientras que para otros verbos las "inmediaciones" conservan su significado original. La forma exacta de definir qué son las inmediaciones para un verbo concreto es complicada, y se sale de los objetivos de este cursillo. Pero que sepas que se puede hacer, y puedes averiguar cómo leyendo el manual DocumentATE, en la sección sobre "el alcance" (DocumentATE llama "alcance" lo que aquí hemos llamado inmediaciones).

También se pueden definir verbos que sólo funcionen sobre un subconjunto de las inmediaciones. Por ejemplo, verbos que sólo funcionan sobre objetos que el jugador tenga en su poder. El verbo EXAMINAR funciona sobre cualquier objeto, pero el verbo DEJAR, sólo debería funcionar sobre las cosas que el jugador lleva consigo. Imagina que quieres en tu juego el verbo EXPRIMIR. Evidentemente, el jugador debe coger un objeto antes de poder exprimirlo. Para forzar a que esto sea así, cuando defines la gramática de ese verbo debes poner la palabra `held` en lugar de la palabra `noun`. La cosa quedará así:

```
Verb 'exprime'
 * held            -> Exprimir;
VerboIrregular "exprimir" with imperativo 'exprime';
[ ExprimirSub;
  "Aprietas ", (el) uno, " pero no sale nada.";
];
```

Ahora, si el jugador lleva consigo el carbón y pone EXPRIMIR CARBON, obtendrá el mensaje "Aprietas el carbón, pero no sale nada." Si el carbón está en otra localidad, obtendrá el mensaje "No veo eso por aquí" (y la acción no se genera). Y el caso interesante es cuando el carbón está en la localidad, pero no en el inventario del jugador. En este caso, el parser se da cuenta de que la acción no puede realizarse, hasta que el jugador tenga el carbón consigo. De modo que emite el mensaje "(primero coges el carbón)" y genera la acción `Coger carbon`. Si esta acción tiene éxito, a continuación genera la acción `Exprimir Carbon`. Fijate que la acción COGER CARBON podría fracasar. Por ejemplo, el carbón podría tener en la rutina `antes` un caso así:

```
  antes [;
       Coger: if (carbon hasnt general)
              "El carbón está todavía demasiado caliente para cogerlo.";
       Soplar: if (carbon has general)
                 "Ya lo has soplado bastante.";
```

```
            give carbon general;
            "Soplas el carbón, enfriándolo.";
],
```

Y en este caso, esto sería lo que vería el jugador si pone EXPRIMIR CARBON:

```
> EXPRIME CARBON
(primero coges el carbón)
El carbón está todavía demasiado caliente para cogerlo.

> SOPLA CARBON
Soplas el carbón, enfriándolo.

> EXPRIME CARBON
(primero coges el carbón)
Aprietas el carbón, pero no sale nada.
```

Es decir, el primer intento de exprimir no llega a producirse, ya que la acción COGER implícita fracasó. En cambio el segundo sí.

Al poner `held` en lugar de `noun` al definir un verbo, dices al parser que el objeto debe estar en posesión del jugador. Si no lo está, el parser genera una acción COGER automáticamente para que lo esté.

Además de `noun` y `held` hay otras palabras especiales que puedes usar al definir una gramática, pero ¡ya está bien para un cursillo! Si quieres saber más, lee el manual DocumentATE.

Ingredientes

Para seguir este tutorial necesitarás las herramientas habituales de desarrollo con Inform, pero eso sí, asegurate de disponer de las últimas versiones de todo ello.

Necesitarás lo siguiente:

- **Librería InformATE! 6/10** (revisión 010515 o superior). El número de revisión hace referencia a la fecha en que fue lanzada. Así 010515 significa año (20)01, mes 05, día 15. Puedes bajar esta librería directamente del archivo de CAAD pinchando aquí (147k)

  También existen versiones "no oficiales", que corrigen algunos bugs. Son no oficiales en el sentido de que los parches no han sido escritos por Zak, sino por Morgul, otro usuario de Inform. Pero aparte de esto, parecen mejor elección que la versión "oficial", ya que como se ha dicho tienen menos bugs. Las versiones no oficiales puedes encontrarlas en la página de Morgul, o si prefieres no buscar, pinchando aqui (153k) te la bajarás directamente del archivo de CAAD.

- **Compilador Informbp 6.21**. Pincha aquí para bajarte del CAAD la versión de Windows, o aqui para el código fuente en C, para que tú mismo lo compiles para tu operativo favorito.
- **Intérprete Frotz**. Con este programa podrás ejecutar los juegos que has creado. Tienes que bajarte la versión apropiada a tu operativo. Por ejemplo, Pulsa aquí para la versión Windows. Tambien tienes una versión DOS, y si usas linux, puedes compilar directamente su código fuente.

Crea una carpeta llamada InformATE y mete dentro esos tres ingredientes. Después descomprime todo ello en esa misma carpeta.

Principios básicos de funcionamiento

¡Hey! ¡He hecho doble click sobre informbp.exe y tan solo se ha abierto una ventana negra y se ha vuelto a cerrar ¿qué ocurre?

Inform no es un "entorno integrado" como Visual Sintac u otros Visuales. Esto quiere decir que no tiene un editor para que puedas escribir tus juegos y desde allí probarlos. Para escribir los juegos necesitas otro programa (el Bloc de Notas de Windows servirá, pero tal vez prefieras usar editores más avanzados). Y para probar los juegos necesitarás otro programa llamado "Intérprete" (El Frotz que te has bajado del enlace anterior es para eso).

Modo de operación:

- Escribe los ejemplos del tutorial usando tu editor favorito (el Bloc de notas servirá). [NOTA: El editor del DOS no sirve!!]
- Antes de abandonar el editor, guarda lo que has escrito en un fichero, en la misma carpeta InformATE donde está todo lo demás. Pongamos que llamas a tu fichero TORRE.INF. Es importante poner .INF como extensión, para acordarte de que es un archivo Inform.
- Abre una ventana del interfaz de comandos (puedes tener una permanentemente abierta para esto), y mediante el comando CD sitúate en la carpeta de InformATE
- Cuando el tutorial te diga que "compiles el juego", se refiere a que debes escribir la siguiente orden en la ventana del interfaz de comandos:

  ```
  INFORMBP TORRE.INF
  ```

  (Si TORRE.INF es el nombre del programa que has escrito)

- Si no ha habido errores, obtendrás un fichero llamado TORRE.Z5 (si los hubo, deberás corregir el programa usando el editor). El fichero TORRE.Z5 es el juego ya listo para ser jugado. Si piensas hacer público tu juego (por favor ¡hazlo!) basta con que distribuyas el .Z5. No necesitas distribuir el .INF (a menos que quieras hacer público tambien el código fuente), ni tampoco necesitas distribuir Frotz puesto que ya casi todo el mundo lo tiene. En todo caso podrás adjuntar en un fichero LEEME.TXT la dirección de Internet donde puede conseguirse Frotz.
- Para probar el juego, lanza WinFrotzE (puedes hacerlo desde el interfaz de comandos o desde el navegador de Windows, como prefieras) y carga en él TORRE.Z5. ¡Eso es todo!