TP - DSLs - Aventuras Gráficas

La idea del TP es construir un lenguaje para desarrollar juegos similares las “aventuras gráficas”, con la pequeñas salvedad de que no vamos a tener una interfaz gráfica :) porque nos agregaría una complejidad bastante grande e innecesaria.
Nos vamos a centrar en el lenguaje, y en la lógica del juego en cuanto a transiciones, acciones, etc.
Entonces más bien es una “aventura conversacional”. Pero tampoco, ja !

Tampoco vamos a poder expresar lógica compleja “custom” de cada juego, porque requeriría que el lenguaje tenga elementos complejos de lenguajes de programación tipo GPL.

Entonces la interacción del juego va a ser por consola, y además (en principio) sólo de lectura. Es decir las acciones del usuario como examinar un objeto, o recogerlo, o ir a una habitación, no se van a ingresar por consola, sino en el lenguaje mismo.

Al “ejecutar” el programa entonces éste va a ejecutar esa lista de “comandos” e ir mostrando por consola lo que sucede como respuesta.

Algunas aclaraciones y/o TIPs

Algunas aclaraciones iniciales antes de arrancar a explicar el dominio del TP, vamos a hablar de algunas cuestiones “de forma”

Lenguaje = Juego + Comandos (input)

Idealmente el DSL contendría únicamente la descripción del juego y sus reglas. Luego al ejecutarlo tendría una consola interactiva, donde el juego iría ejecutándose por pasos, interactuando con el usuario. Es decir que el usuario ingresaría texto por la linea de comandos, y esos se ejecutarían. Y luego ingresaría otro, etc, etc..

Podrían implementarlo así, pero la manera más “manual” de implementarlo es leyendo por consola y esos “comandos” del usuario no serían parte de la especificación del lenguaje. Así que vamos a evitar eso.

Entonces vamos a hacer que el lenguaje incluya en el mismo archivo tanto la descripción del juego como también los comandos que ingresaría el usuario para jugar.
Sí, es un poco “precario”, pero es la forma más simple de evitar parsear strings de consola, etc.

En general los “comandos” producen mensajes de salida que irán a la consola.

El Intérprete

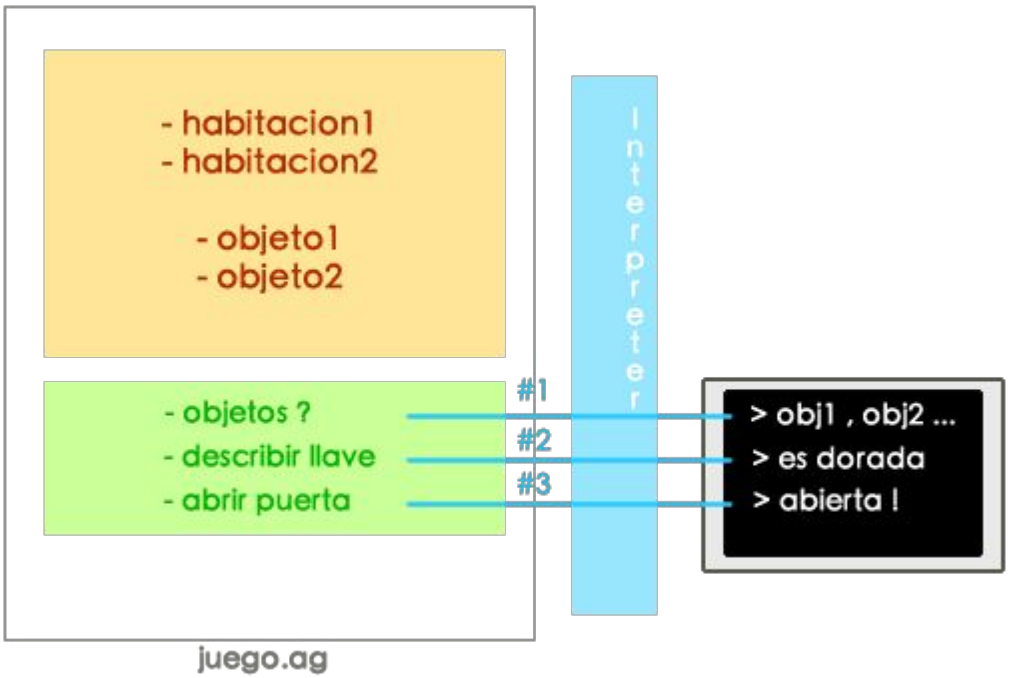
El lenguaje entonces será “ejecutable”. Al programa que lo ejecutará le decimos “intérprete”.
Tenemos algunos ejemplos en el site de la materia así como también en el SVN (ejempo de “saludos” y el ejemplo de “tortuga”)
Más data en este link

Y aca un esqueleto completo de la clase del interprete que va a venir bien para no arrancar de cero.

El intérprete es un programa con un main que va a recibir como argumento el path completo a un archivo escrito en sus lenguajes.
Deberá usar la API de xtext para parsearlo y validarlo. Al final de esto van a tener un objeto instancia de sus modelo semántico (las clases que les generó XText en base a su gramática). Y ahí comienza la diversión.

Deberán ir “interpretando” cada objeto y determinando qué hacer.
Como no podemos meterle el comportamiento a esas clases generadas, solemos usar extension methods para eso.

En este caso, el intérprete lo que debe hacer es evaluar cada uno de los comandos, que es la parte que realiza un comportamiento. Y eso posiblemente genere salida en la consola.



Dominio

El dominio del juego consta de habitaciones y objetos.
El usuario siempre está en una única habitación en un momento dado.
Las habitaciones tienen dentro objetos.
Los objetos pueden tener una descripción, que sirve para dar algún tipo de pista al usuario sobre su uso, etc.
Los objetos también tienen acciones que el usuario puede ejecutar sobre el mismo.

Luego aparecen algunas variantes y conceptos adicionales que vamos a ir viendo de a poco en cada iteración.

Recorrido para la implementación

A modo de guía planteamos acá un par de ejemplos que forman un camino incremental para ir implementando el lenguaje en forma iterativa.

Parte 1

Primera parte del TP. Contiene trabajo en todas los aspectos del lenguaje (gramática, checkeos e intérprete) pero con funcionalidad simple/acotada.

Iteración 0: habitación inicial

Poder describir una habitación vacía. Se espera que al ejecutar el intérprete se muestre el mensaje de bienvenida a la habitación inicial

```
objeto llave
objeto tapete
objeto puerta
objeto auriculares

iniciar en fueraDeLaMansion

habitacion fueraDeLaMansion
      al ingresar
            mostrar "Estas en las afueras de la mansión Edison. Tienes que
entrar a rescatar a tu amigo Bernard"

habitacion lobby
      al ingresar
            mostrar "Pum Pum... Se escucha un ruido de la planta alta"
```

Al ejecutar esto, si bien no hay ningún comando deberíamos ver:

```
> Estas en las afueras de la mansión Edison. Tienes que entrar a rescatar a tu
amigo Bernard
```

Iteración 1: objetos y descripciones

Los objetos ahora “pueden” tener una descripción, y existen dos comandos:

- **Mostrar Objetos:** para preguntar por los objetos que existen en la habitación actual
- **Describir Objeto “X”:** que nos da detalles sobre el objeto X.

Acá un ejemplo extendiendo el juego anterior:

```
objeto llave
      descripcion "Una llave vieja. Parece de una puerta"
objeto tapete
objeto puerta
objeto auriculares


iniciar en fueraDeLaMansion

habitacion fueraDeLaMansion
      al ingresar
            mostrar "Estas en las afueras de la mansión Edison. Tienes que
entrar a rescatar a tu amigo Bernard"
      objetos
            - puerta
            - tapete
            - llave
// comandos
objetos ?
describir llave
describir puerta
```

Produce:

```
> Estas en las afueras de la mansión Edison. Tienes que entrar a rescatar a tu
amigo Bernard
$ objetos ?
> Hay: puerta, tapete, llave
$ describir llave
> Una llave vieja. Parece de una puerta
$ describir puerta
> No puedo decir nada sobre puerta
```

Tip:

En este ejemplo, el programa está imprimiendo no sólo las respuestas (acá marcado en azul) sino también el texto original de los comandos (en negro).
Eso es posible porque en XText, dado un objeto del modelo semántico podemos obtener su “Nodo” del AST (o sea un objeto que representa el elemento sintáctico. Piénsese como un objeto de más bajo nivel, más cerca del “texto”). Esto se hace así:

```
val node = NodeModelUtils.findActualNodeFor(modelo)
```

```
val texto = node.text()
```

Ó mejor, como es un método static lo podemos usar como una extension

```
import static extension org.eclipse.xtext.nodemodel.util.NodeModelUtils.*

val texto = modelo.findActualNodeFor.text()
```

Iteración 2: Acciones de objetos

Los objetos pueden tener acciones que luego se pueden “consultar”, y “ejecutar”.
En este primer caso vamos a tener un único “tipo de acción” o comportamiento: ir a otra una habitación

```
objeto puerta
	descripcion "No parece estar cerrada"
	accion entrar hace ir a lobbyDeLaMansion

iniciar en fueraDeLaMansion

habitacion fueraDeLaMansion
	al ingresar
		mostrar "Estas en las afueras de la mansión Edison. Tienes que
entrar a rescatar a tu amigo Bernard"
	objetos
		- puerta

habitacion lobbyDeLaMansion
	al ingresar
		mostrar "Es un lugar viejo. Huele a húmedo"

// comandos
objetos ?
describir puerta
acciones puerta ?
accion entrar a la puerta
```

Esto va a generar:

```
Estas en las afueras de la mansión Edison. Tienes que entrar a rescatar a tu
amigo Bernard
$ objetos ?
Hay: puerta
$ describir puerta
```

```
No parece estar cerrada
$ acciones puerta ?
Se puede: entrar
$ accion entrar a la puerta
Es un lugar viejo. Huele a húmedo
```

Tip: El nuevo comando “accion entrar a la puerta” consta de dos referencias, la primera es a una regla de tipo “acción” es decir a algo que se puede hacer con un objeto que tiene que estar definido en el objeto “puerta” en esta caso, que es la segunda referencia. O sea

`‘accion’ Accion ‘a la’ Objeto`

El tema es que XText se va a quejar con que no encuentra la acción “entrar”, no la puede resolver.

Esto se debe a que ese objeto “entrar” está definido “dentro” del objeto “puerta”. Xtext busca referencias automáticamente, pero en elementos “hermanos” o que estén en un nodo padre. Ya que el texto se puede ver en realidad como un árbol.

Desde “acción” entonces podemos ver al Juego, podemos ver “puerta” y “habitación”, pero, no vemos los elementos que estén **dentro** de puerta o de habitación.

A la visibilidad de las referencias se la llama “scoping”. XText ya provee este scoping default jerárquico, pero podemos customizarlo.
Para eso hay que modificar la clase:

```
<<NombreDelLenguaje>>ScopeProvider
```

Esta clase trabaja con una convención de nombres de métodos.

Primero veamos un ejemplo de cómo sería la regla para “Ejecutar acción”

```
EjecutarAccion: 'accion' accion=[Accion] ('a'|'al'|'a la')? objeto=[Objeto];
```

El elemento que nos interesa ahí es el marca en negrita y subrayado. La propiedad “accion” de tipo Accion es la que queremos customizar. Ese scope, es decir los elementos que van a estar visible en ese punto.
Para eso tenemos que definir un método en el scope provider con esta forma:

```
def IScope scope_Accion(EjecutarAccion ctx, EReference ref) { ... }
```

Los dos elementos importantes son los marcados en negrita.
Esto se lee como “el scope al resolver un objeto de tipo **Accion**, desde un objeto de tipo **EjectuarAccion**” se computa con este método.

Fijense que “EjecutarAccion” es el nombre de nuestra regla y Acción el nombre de la que queremos referenciar.
Tambíen fíjense que el método recibe al objeto EjecutarAcción que es justo lo que necesitamos.
Porque qué acciones le vamos a dejar escribir al usuario ? Las que pertenezcan al objeto.
En este caso sólo podrá especificar una acción de “puerta”.

Entonces la implementación queda así:

```
class NaturalScopeProvider extends AbstractDeclarativeScopeProvider {

	def IScope scope_Accion(EjecutarAccion ctx, EReference ref) {
		val accionesVisibles = ctx.objeto.acciones
		return crearScope(accionesVisibles)
	}

	def crearScope(List<Accion> acciones) {
		new SimpleScope(acciones.map[
			EObjectDescription.create(QualifiedName.create(name), it)
		])
	}

}
```

Tiene un poco de código específico de XText para crear el scope, pero lo importante es “`ctx.objeto.acciones`”.

Con esto el ejemplo debería compilar y funcionar.

Parte 2

Contiene comportamiento más avanzado/complejo como manejo de estado.

Iteración 3: Estado en los objetos + des/habilitar acciones

Vamos a agregarle capacidad de habilitar y deshabilitar acciones de los objetos, en base a un estado.
Los objetos ahora pueden tener algo similar a “variables de instancia”. Por ahora el tipo de los valores va a ser STRING, no queremos complicarlo demasiado por ese lado.
Además necesitamos

- poder definir una condición en la cual una acción está habilitada en base al valor de una variable del objeto
- poder modificar ese valor como resultado de ejecutar una acción

Veamos un ejemplo donde la puerta se puede abrir sólo si está cerrada, y se puede entrar a ella, sólo cuando está abierta.
Deshabilitar una acción significa que:

- no se va a mostrar cuando ejecutemos el comando de “mostrar acciones”
- si igualmente intentamos ejecutarla, nos va a decir un mensaje como “acción no disponible”. Incluso sería muy groso que en este ejemplo dijera “Acción no disponible ya que el estado de la puerta es abierta y debería ser cerrada” (claro sin hardcodear el mensaje sino generandolo en forma genérica)

```
objeto puerta
	descripcion "No parece estar cerrada"
	tiene estado con valor "cerrada"

	accion abrir hace cambiar el estado al valor "abierta"
		cuando estado es "cerrada"
	accion cerrar hace cambiar el estado al valor "cerrada"
		cuando estado es "abierta"

	accion entrar
		hace ir a lobbyDeLaMansion
		cuando estado es "abierta"
```

“con valor …” es opcional. Es decir una variable podría no tener un valor inicial

La parte restante del ejemplo sería:

```
iniciar en fueraDeLaMansion

habitacion fueraDeLaMansion
      al ingresar
            mostrar "Estas en las afueras de la mansión Edison. Tienes que
entrar a rescatar a tu amigo Bernard"
      objetos
            - puerta


habitacion lobbyDeLaMansion
      al ingresar mostrar "Bienvenido al lobby de la mansion"


// comandos
acciones puerta ?

accion abrir puerta
acciones puerta ?

accion cerrar puerta
acciones puerta ?

accion abrir puerta
accion entrar puerta
```

Y al ejecutarlo produce algo así:

```
Estas en las afueras de la mansión Edison. Tienes que entrar a rescatar a tu
amigo Bernard
$ acciones puerta ?
Se puede: abrir
$ accion abrir puerta
$ acciones puerta ?
Se puede: cerrar, entrar
$ accion cerrar puerta
$ acciones puerta ?
Se puede: abrir
$ accion abrir puerta
$ accion entrar puerta
Bienvenido al lobby de la mansion
```

Tip:

El intérprete ahora necesita mantener estado, los “valores actuales” de cada variable de objeto. Uds ya van a tener un objeto VariableDeObjeto (o el nombre que le den) que va a ser una regla de la gramática y entonces una clase del modelo semántico, sin embargo como son clases generadas no podemos modificarla para meterle un “valor”. Y tampoco es fácil modificarle el “valor inicial” (también parte de la gramática).

Entonces lo más simple que el estado de las variables lo maneje el intérprete. De alguna forma debe guardarse un mapeo

```
VariableDeObjeto -> ValorActual (STRING)
```

Obtener el valor actual, por ejemplo para saber si un acción está disponible o no, significa ir a buscarlo ahí (o si no está usar el inicial del objeto mismo). Modificar el valor (por ejemplo como resultado de una acción) es cambiar el valor en ese mapeo.

Iteración 4: Agregar Checkeos/Validaciones

- No se pueden declarar dos objetos con el mismo nombre
- No se pueden declarar dos habitaciones con el mismo nombre
- Las habitaciones deben tener al menos un objeto con una acción que permita ir a otra habitación (de otra forma no tendrían salida). Salvo que sea una “habitación final” (requiere modificar la gramática para de alguna forma declarar esto).

Iteración 5: Inventario

Agregar la posibilidad de recoger algunos objetos y guardarlos en el inventario.
Esto involucra:

- Poder definir cuales son los objetos que “se pueden recoger” a la hora de definir el objeto mismo.
- Agregar el comando “recoger objetoA” que:
    - si el objeto se puede recoger se lo guarda en el inventario (y dice algo avisando que pudo)
    - si el objeto no se puede recoger avisa de ésto con otro mensaje.
    - si ya tiene el objeto en el inventario no hace nada, pero también le avisa al usuario que ya lo tenía.
- Agregar el comando “inventario ?” que muestra el contenido actual del inventario.
- Otros impactos del cambio:
    - “objetos ?” ahora debería mostrar sólo los objetos de la habitación actual que no fueron recogidos por el usuario (que no están en el inventario)
    - “acciones objetoA ?” (es decir consultar por las acciones posibles sobre un objeto) deberá incluir el “recoger” en caso en el que aplique (objeto es recogible y además aún no fue recogido).

Bonus 1: Modificar objetos al ingresar a habitación

Hasta ahora podíamos definir un único comportamiento al ingresar a una habitación que era el mostrar un mensaje.

Deben agregar otra posibilidad: modificar el estado de un objeto al ingresar.
Por ejemplo para modelar el siguiente caso:

```
habitacion lobbyDeLaMansion
      al ingresar
                - mostrar "Entras caminando lentamente y la puerta se cierra
                  detrás tuyo"
                - cambiar el estado de puerta al valor "cerrada"
```

“estado” es una variable del objeto “puerta” que debe estar incluído en la lista de objetos de esta habitación (checkeo !)

Bonus 2: Usar objetos

Pensar e implementar un conjunto de comandos y gramática para utilizar un objeto con otro.
Por ejemplo usar la llave para abrir la puerta de la habitación.