Trabajo Práctico: Diseño e Implementación de un Lenguaje

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Primer Cuatrimestre 2025

Grupo G-72

Ana Negre - 63074

Matías Leporini - 62872

Tabla de contenidos

1. Introducción

A lo largo de las tres entregas del presente trabajo práctico, se buscó diseñar y construir un *Domain Specific Language* para la generación de presentaciones, además de la implementación de un compilador para el mismo. Para lograrlo, se hizo uso y expandió un compilador de Flex-Bison provisto por la cátedra. Concretamente, se optó por un lenguaje orientado a la creación de presentaciones para simplificar el proceso de creación de filminas, tanto en cantidad de tiempo requerido como en términos de complejidad del programa. Adicionalmente, se buscó evitar la necesidad de utilizar aplicaciones especializadas para la visualización de las presentaciones, por lo que el producto final de compilación provee un archivo *HTML*, el cual puede ser visualizado en cualquier navegador web local.

Este lenguaje, "*Pressed*", provee las funcionalidades fundamentales de plataformas del género (como *PowerPoint* o *Canva*), pero sin la necesidad de utilizar una interfaz gráfica para ello. Algunas de las posibilidades que ofrece son la inserción de texto e imágenes y su customización, animaciones y posicionamiento relativo de objetos. Una de las grandes ventajas que otorga frente a alternativas es la posibilidad de definir objetos para reutilizar con distintos valores a lo largo de la presentación; como ejemplo, puede definirse un bloque de texto con tamaño de letra y color específicos, el cual podrá ser utilizado como título en cada diapositiva con un *texto* diferente en cada ocasión.

2. Modelo computacional

2.1. Dominio

Pressed permite crear presentaciones mediante componentes simples como *Slide*, *Textblock* e *Image*. De esta forma, habilita al usuario a definir filminas, establecer su contenido, color y otras propiedades. También brinda la posibilidad de determinar la disposición de los objetos en cada filmina, ubicándolos por defecto en el orden de añadidura, pudiendo utilizar sentencias de posicionamiento relativo para personalizar la ubicación. Por último, permite la adición de animaciones mediante una conjunción de estructuras de control y otro set de operadores. La salida del lenguaje es un archivo HTML, simulando así una presentación de *PowerPoint*, pero con el beneficio de ser considerablemente más liviana y de fácil acceso (ya que cualquier navegador puede abrir el archivo).

2.2. Lenguaje

Todo código en *Pressed* se divide en tres secciones: `Objects`, `Structure` y `Animations`. En la primera se declaran los **objetos** a utilizarse en la presentación, junto a sus propiedades. Aquí se pueden determinar valores como el tamaño del texto, el ancho máximo de una imagen, etc. En segundo lugar, se encuentra la **estructura**. Aquí se determina qué objetos se encuentran en cada filmina, con qué valores (texto o URL de la imagen) y cómo están posicionados. Los objetos son reutilizables, por lo que pueden agregarse en más de una filmina. Por último, en el apartado final se pueden definir bloques de **animaciones** para cada diapositiva, otorgando la posibilidad de repetir una secuencia de animaciones un cierto número de veces. Además, existe la posibilidad de agregar una animación de transición entre las filminas. Cabe destacar que las tres secciones están englobadas en una sección de nombre `Presentation`, siendo necesario otorgarle un nombre específico a la misma, que funcionará como nombre del archivo del output. A continuación se desarrollará brevemente a cada una de las tres secciones internas.

Para empezar, en `Objects` se definen objetos del tipo `Slide`, `Textblock` e `Image`. Esto se hace con la sintaxis `name ObjectType { properties }`, siendo cada propiedad un par nombre-valor de la forma `property-name : value;`. Cada diapositiva, bloque de texto e imagen debe definirse en esta sección para ser utilizable más adelante, incluso si no se lo customizara. Las propiedades disponibles son:

Para `Image`:

+ `border-size`
+ `border-color`
+ `border-style`
+ `border-radius`
+ `max-height`
+ `max-width`

Para `Textblock`:

+ `background-color`
+ `font-size`
+ `font-family`
+ `font-color`

Para `Slide`:

+ `background-color`
+ `font-family`

Todas surgen de las propiedades CSS utilizadas internamente por el compilador para construir la presentación, por lo que los valores aceptados son aquellos que forman parte tanto de CSS como del alfabeto (por ejemplo, `12px`, `rgb(10,10,10)`, `sans-serif`, etc).

`Structure` se separa internamente por diapositiva; todos los objetos dentro de cierta filmina deberán colocarse en el bloque `slideName { }`. Dentro de cada uno de estos bloques, pueden agregarse objetos ya definidos mediante `add objectName with`

`value;`, siendo el valor un *string* en el caso de estar siendo aplicado a texto y un *path* (también en formato de *string*) en caso de estar siendo aplicado a una imagen. Por último, se puede dictaminar la disposición relativa de dos objetos mediante `objectName1 relativePosition objectName2;`. Las sentencias de posicionamiento deben estar luego de la declaración de objetos pertenecientes a la filmina. El operador binario de posición puede tomar el valor de cualquier dirección cardinal, además de sus permutaciones. Cabe destacar que las diapositivas se ubicarán en el documento HTML de salida por orden de aparición en esta sección.

Por último se encuentra `Animations`. Allí puede haber un bloque de animaciones por cada filmina, el cual es de la forma `slideName start then objectName1 animation then objectName2 animation2 then ... end repeat times;`. La animación puede ser `rotate`, `appear` o `disappear`, y el fragmento que permite repetir todas las animaciones es opcional. Si existiese más de una secuencia de animaciones para una filmina, se tomará únicamente la última secuencia definida. Para finalizar, es posible agregar una animación entre dos diapositivas de la forma `animation slideName;`. Cabe destacar que únicamente se usa el nombre de la diapositiva sobre cuya aparición se aplicará la animación de transición, y que es posible agregarle una transición a la primera filmina (ya que, al volver atrás en la presentación, las animaciones se pasan al revés). En este caso, la animación puede ser `fade_into` o `jump_into`.

3. Instrucciones de uso

Inicialización del contenedor en *Linux*:

```
./script/ubuntu/docker-build.sh
./script/ubuntu/docker-run.sh
```

Inicialización del contenedor en *Windows*:

```
./script/windows/docker-build.bat
./script/windows/docker-run.bat
```

Dentro del contenedor:

```
./script/ubuntu/install.sh
./script/ubuntu/build.sh
```

Para correr los *tests*:

```
./script/ubuntu/test.sh
```

Para compilar un programa de *Pressed*:

```
./script/ubuntu/start.sh pathAlPrograma
```

Opcionalmente, se puede agregar *-v* para modo verboso y ver los logs hasta nivel *DEBUG*:

```
./script/ubuntu/start.sh pathAlPrograma -v
```

Para ver la presentación resultante, se puede abrir el HTML producido en la carpeta *output* desde cualquier navegador o visualizador de HTML que soporte *JavaScript*. Para moverse entre diapositivas o animaciones, se utilizan las flechas izquierda y derecha.

4. Implementación

4.1. Frontend

El *frontend* del proyecto respecta a dos áreas: el análisis léxico y el análisis sintáctico. Comenzando por la parte léxica, en el archivo ***FlexPatterns.l*** se definieron los patrones que serán reconocidos como *tokens*. Entre los mismos se encuentran aquellos relacionados a los tipos de objetos propios del lenguaje (`Slide`, `Image`), animaciones (`appear`, `fade`), posicionamiento (`above`, `left`) y otros. Todos los lexemas recién listados son *keywords*, pero también se definieron patrones para reconocer números enteros, decimales, *strings*, corchetes y comentarios. Continuando con el flujo, en ***FlexActions.c*** se determinan los pasos a seguir según el *token* reconocido (que podría ser un literal, una propiedad o un símbolo), tomando las funciones formas similares a la siguiente:

```
Token IdentifierLexemeAction(LexicalAnalyzerContext * lexicalAnalyzerContext) {
	_logLexicalAnalyzerContext(__FUNCTION__, lexicalAnalyzerContext);
	lexicalAnalyzerContext->semanticValue->string =
				strdup(lexicalAnalyzerContext->lexeme);
	destroyLexicalAnalyzerContext(lexicalAnalyzerContext);
	return IDENTIFIER;
}
```

Finalizando con el analizador léxico, en ***LexicalAnalyzerContext.c*** y su archivo header se ofrecen las funciones requeridas para crear y destruir objetos de contexto, además de definirse el *struct* encargado de mantener a cada uno.

En cuanto al analizador sintáctico, en ***BisonGrammar.y*** se define la estructura de un programa de *Pressed*, sobre la cual se ahondará en mayor detalle:

```
program:
```

```
      PRESENTATION IDENTIFIER OPEN_CURLY_BRACE objects structure animation
                                                          CLOSE_CURLY_BRACE
      | PRESENTATION IDENTIFIER OPEN_CURLY_BRACE objects structure
                                                  CLOSE_CURLY_BRACE
```

Lo que está determinando este fragmento de código es que todo programa inicia con la *keyword* `Presentation`, seguida por corchetes. Dentro de estos corchetes van las secciones de objetos, estructura y -opcionalmente- animaciones.

`Objects` toma la siguiente forma:

```
objects:
      OBJECT OPEN_CURLY_BRACE object_definitions CLOSE_CURLY_BRACE

object_definitions:
      %empty | object_definitions object_definition

object_definition:
      SLIDE IDENTIFIER OPEN_CURLY_BRACE css_properties CLOSE_CURLY_BRACE
      | TEXTBLOCK IDENTIFIER OPEN_CURLY_BRACE css_properties CLOSE_CURLY_BRACE
      | IMAGE IDENTIFIER OPEN_CURLY_BRACE css_properties CLOSE_CURLY_BRACE
```

Así, se indica que en esta sección se definirán propiedades para diapositivas, bloques de texto e imágenes.

En `Structure`, se define qué objetos van en cada diapositiva (`slide_contents`) y el posicionamiento de los mismos (`position_contents`):

```
structure:
      STRUCTURE OPEN_CURLY_BRACE structure_definitions CLOSE_CURLY_BRACE

structure_definitions:
      %empty | structure_definitions structure_definition

structure_definition:
      IDENTIFIER OPEN_CURLY_BRACE slide_contents position_contents
CLOSE_CURLY_BRACE

slide_contents:
    %empty | slide_contents slide_content

slide_content:
    ADD IDENTIFIER SEMICOLON | ADD IDENTIFIER WITH STRING SEMICOLON

position_contents:
```

```
    %empty | position_contents position_content

position_content:
    IDENTIFIER simple_position IDENTIFIER SEMICOLON
    | IDENTIFIER compound_position IDENTIFIER SEMICOLON
```

Por último, en `Animations` se especifican las posibles estructuras de un bloque de animaciones; se coloca el identificador de la diapositiva a animar, y se puede elegir si iniciar una secuencia de animaciones (que opcionalmente puede repetirse) o si agregarle una transición a una diapositiva.

```
animation:
    ANIMATION OPEN_CURLY_BRACE animation_definitions CLOSE_CURLY_BRACE

animation_definitions:
    %empty | animation_definitions animation_definition

animation_definition:
    animation_type_slides IDENTIFIER SEMICOLON
    | IDENTIFIER START animation_sequence END SEMICOLON
    | IDENTIFIER START animation_sequence END REPEAT INTEGER SEMICOLON

animation_sequence:
    %empty | animation_sequence animation_step

animation_step:
    THEN IDENTIFIER animation_type
```

En este breve recorrido por el archivo se obviaron tanto las definiciones consideradas sencillas (lista de animaciones, operadores binarios de posicionamiento) como las acciones de *Bison* llamadas por cada expresión. Respecto a esto último, en ***BisonActions.c*** se determina la acción a tomar según cada regla de la gramática para la construcción del árbol sintáctico. En el ejemplo a continuación se crea una transición para cierta diapositiva, generando la estructura correspondiente y haciendo el chequeo semántico necesario.

```
AnimationDefinition *AnimationDefinitionPairSemanticAction(CompilerState
*CompilerState, char *id1, AnimationType type) {
      _logSyntacticAnalyzerAction(__FUNCTION__);
      SymbolTableItem *slideItem1 = getSymbol(CompilerState->symbolTable, id1);
      if (slideItem1 == NULL || slideItem1->type != OBJ_SLIDE) {
            logError(_logger, "Object with identifier '%s' does not exist or is
                              not a slide.", id1);
            CompilerState->errorCount++;
```

```
    }
    AnimationDefinition *animation = calloc(1, sizeof(AnimationDefinition));
    animation->kind = ANIM_DEF_PAIR;
    animation->pair.identifier1 = id1;
    animation->pair.type = type;
    return animation;
}
```

Dentro de ***AbstractSyntaxTree.h*** se definen los *structs* a ser utilizados en el AST. En el denominado `AnimationDefinition` se ven representadas las dos posibles alternativas para una estructura de animación (ya vistas en ***BisonGrammar.y***), entre las cuales se hace una unión. Cabe destacar que `AnimationStep` es una *linked list* de los pasos en una secuencia de animaciones.

```
typedef struct AnimationDefinition {
    AnimationDefinitionKind kind;
    union {
        struct {
            char *identifier;
            AnimationStep *steps;
            int repeat_count;
        } sequence;
        struct {
            char *identifier1;
            AnimationType type;
        } pair;
    };
    struct AnimationDefinition *next;
} AnimationDefinition;
```

Ya llegando al fin del *frontend*, en ***AbstractSyntaxTree.c*** existen métodos para liberar de la memoria todas las estructuras definidas, y en ***SyntacticAnalyzer.c*** se deriva el parseo a *Bison*, pudiendo consecuentemente aceptar o rechazar el código. Además, se ofrecen estados específicos como `OUT_OF_MEMORY` y `UNKNOWN_ERROR`, visibles junto al resto mediante la estructura `CompilerState`.

4.2. Backend

Para el *backend*, se utiliza la estructura de estado del compilador para almacenar la tabla de símbolos, (además del AST resultante del *frontend*) y la posterior generación del *layout* de los elementos de la presentación . El mismo tiene la siguiente estructura:

```
typedef struct {
	void *abstractSyntaxtTree;
	boolean succeed;
	int errorCount;
	int slideCounter;
	SymbolTable *symbolTable;
	SlideList *slides;
} CompilerState;
```

La tabla de símbolos se construye al mismo tiempo que se construye el AST durante el *parsing*, puesto a que permite realizar los chequeos semánticos pertinentes (redeclaraciones de variables, utilización incorrecta de variables según tipo, etc.), los cuales no pueden ser detectados en la etapa de análisis sintáctico de *frontend*. En caso de encontrar errores semánticos, se permite que se genere el AST, teniendo un contador para los mismos, pero no se procede a la posterior etapa de generación del output. Esto se decidió para garantizar la máxima integridad posible del programa de input en la fase de generación de código.

Respecto a la recién mencionada tabla de símbolos, ***SymbolTable.h*** incluye dos estructuras de relevancia. La estructura padre (`SymbolTable`) está formada por una *HashTable* con pares { identificador, `SymbolTableItem` }, y la estructura hijo contiene:

- `type`: Previsiblemente, el tipo del objeto (`OBJ_SLIDE`, `OBJ_TEXTBLOCK`, `OBJ_IMAGE`).
- `currentSlide`: Si el objeto es una diapositiva, funciona como identificador numérico. En otro caso, es el identificador de la última diapositiva en la que fue encontrado el objeto, funcionando como *scope*.
- `appearsIn`: *Array* de diapositivas en las que aparece el objeto. Se decidió implementar como *array* dado que no es importante el orden en que aparecen (obviando la necesidad de un stack)
- `properties`: Puntero a su lista de propiedades del objeto.

```
typedef struct {
	ObjectType type;
	int currentSlide;
	GArray *appearsIn;
	CssProperty *properties;
} SymbolTableItem;

typedef struct {
	GHashTable *table;
} SymbolTable;
```

También se define la estructura `SlideList`, una lista formada por un conjunto de `Slide`. Este último *struct* contiene:

- `rows`: *HashTable* que mapea índices de fila a estructuras `Row`. Las mismas contienen otra *HashTable* responsable de administrar las columnas de la presentación (con sus respectivos indicadores `minCol`, `maxCol`)
- `minRow`, `maxRow`: Límites de la cuadrícula ocupada por los objetos.
- `symbolToObject`: *HashTable* que mapea identificadores a objetos posicionados.

```
typedef struct Slide {
	char *identifier;
	GHashTable *rows;
	int minRow;
	int maxRow;
	int minCol;
	int maxCol;
	GHashTable *symbolToObject;
	struct Slide *next;
} Slide;

typedef struct {
	Slide *head;
	Slide *tail;
} SlideList;
```

La estructura `PositionedObject` permite realizar las búsquedas inversas para el reposicionamiento de los objetos a partir de las sentencias de posicionamiento.

```
typedef struct PositionedObject {
	char *identifier;
	int row;
	int col;
} PositionedObject;
```

El archivo ***layout.c*** se encarga de posicionar todos los objetos en una cuadrícula dinámica, utilizando las estructuras recién desarrolladas. Debido a que esta sección sufrió una variedad de cambios y reestructuraciones, se tratará en mayor detalle en el fragmento de dificultades encontradas.

Por otro lado, ***properties.c*** específicamente maneja las propiedades CSS de todos los objetos. dividiendo el parsing por tipo de objeto (`Slide`, `Textblock`, `Image`).

Respecto a la generación de código, se divide en dos partes: HTML y CSS. De ambas se encarga ***generator.c***; primero crea un archivo con el nombre que el usuario le dio a la presentación en un directorio "*output*", y le concatena las líneas de HTML que se asocian al código creado. Además, se agregan un prólogo (con necesidades básicas de HTML y una referencia a un archivo CSS con propiedades por defecto) y un epílogo (con el número de diapositiva y referencias a todos los programas de *JavaScript* utilizados).

En cuanto al estilo, la función `outputProperties` concatena las propiedades elegidas por el usuario dentro del fragmento `<style>` en el prólogo. Las propiedades no soportadas por el lenguaje desencadenan un error, sino que simplemente no se incluyen en el resultado final.

En el proyecto, se incluye un número de archivos *JavaScript* orientados a proveer el funcionamiento de la presentación:

- *main.js*: Inicializa el core de la aplicación cuando carga el DOM.
- *core.js*: Inicializa y llama al resto de los módulos.
- *controller.js*: Permite avanzar o retroceder en la presentación (tarea que en parte se deriva a *slides.js*), garantizando que los eventos transcurran en el orden esperado. Lee *inputs* de las flechas laterales del teclado.
- *slides.js*: Se encarga de determinar la diapositiva actual, almacenar un historial de diapositivas y un historial de estados. Estas dos estructuras se usan para permitir volver hacia atrás en la presentación.
- *ui.js*: Provee soporte para el número de diapositiva (visible en la esquina inferior derecha).
- *animation.js*: Determina animaciones individuales para objetos en diapositivas.
- *animation-sequence.js*: Permite navegar entre animaciones, incluyendo los casos en los que un objeto tiene una secuencia de las mismas
- *transition.js*: Provee el funcionamiento para transiciones entre diapositivas.

Además, se creó un archivo *styles.css* con estilos no modificables y los valores por defecto de cada componente, junto a las animaciones de los objetos y las transiciones.

4.3. Dificultades encontradas

En primer lugar, incluso antes de iniciar con el desarrollo, fue complejo definir el lenguaje en sí; las presentaciones son entidades complejas, y encontrar una forma coherente y eficiente de determinar propiedades, posiciones y animaciones tomó una cantidad notable de deliberación. En particular, algunas nociones de cómo ejecutar el posicionamiento relativo recién terminaron de ser formadas en el transcurso de la tercera entrega, requiriendo cambios estructurales considerables.

Estas complejidades se manifestaron a la hora de programar el *backend* para el posicionamiento relativo; dado que resultaba imposible considerar las posiciones de todos los objetos al mismo tiempo que se construía el AST -ya que no se contaría con toda la información al respecto hasta el final- la disposición debía definirse posteriormente. En base a eso, se decidió que una vez construido el AST, se definiría el *layout* mediante una matriz de posiciones. Para hacer a este concepto más eficiente, dentro de cada nodo de la lista de estructuras se separó la lista de sentencias de posicionamiento de la lista de sentencias de adiciones a una diapositiva (`add` image y similar). Además, se estableció una nueva limitación en la gramática: las sentencias de posicionamiento deben ser posteriores a las adiciones de elementos.

Continuando con el tema, dado que el posicionamiento es relativo, la ubicación de un objeto hijo debe resolverse en base a la locación del padre, agregando carácter recursivo a la solución. Sumado a esto, durante el recorrido de la lista de sentencias de posicionamiento, una vez resuelta la ubicación de un objeto, el mismo está sujeto a cambios en caso de encontrar otras sentencias que lo involucren. Por esta razón, y para hacer más eficiente las búsquedas (puesto a que se debe parsear la matriz de posiciones y rearmarse al encontrarse nuevos) se decidió utilizar un *HashMap* con pares { índice de fila o columna, identificador del objeto }. Adicionalmente, se utilizó otro *HashMap* para búsquedas inversas. Es decir, para averiguar una posición según un identificador. En caso de haber conflictos, se decidió resolverlos de la siguiente manera:

En caso de que un objeto fuera hijo en más de una sentencia, se toma solo la primera sentencia encontrada, descartando las demás. Además, para evaluar las dependencias de posicionamiento, se decidió utilizar un mapa para detectar ciclos y resolver recursivamente la ubicación del padre. Se destaca que el posicionamiento puede ser mejorado en varios aspectos, en especial para eliminar los conflictos entre sentencias que compartan hijos.

Un cambio implementado luego del desarrollo del *frontend* (el cual significó modificaciones en el mismo), fue la eliminación del *anchor* para las presentaciones. Se consideró que, a fines prácticos, solo funcionaba como un objeto más que no aportaba mucho valor al lenguaje, dado que en la abrumadora mayoría de los casos se utilizaría el anchor en el centro superior de la página. Por ende, se determinó que el "ancla" siempre estará en la parte superior de la diapositiva, y las columnas de cada fila quedarán centradas. Eliminar el concepto de *anchor* permitió simplificar la generación del layout, si bien se considera que su eventual implementación no debería ser priorizada por encima de la inclusión de posicionamiento absoluto (ver sección de futuros avances).

Una última complicación encontrada en el desarrollo fue la complejidad de los archivos *JavaScript* requeridos para que la presentación funcione de manera adecuada. Gracias a una serie de restricciones autoimpuestas fue necesario crear una cantidad de funciones y estructuras que originalmente no se habían considerado. En particular, dado que se pretendía que -al volver hacia atrás en la presentación- las animaciones y transiciones volvieran a ejecutarse en orden inverso, se agregó una estructura para almacenar los estados de las diapositivas a lo largo del documento. Esto condicionó de forma notable al resto de las funcionalidades de *JavaScript*, pero se considera que valió la pena: habilita una experiencia pulida del lado del usuario.

5. Futuras extensiones

No es de extrañar que, dada la enorme cantidad de funcionalidades que otorga *PowerPoint* -la inspiración de *Pressed*- un considerable número de posibles adiciones al lenguaje hayan permanecido en el tintero. Por ejemplo, desde un inicio se barajó la posibilidad de ofrecer disposiciones predeterminadas para las filminas, evitando así que el usuario deba ubicar los objetos de forma relativa. Sin embargo, debido a que el tiempo disponible no hacía viable la inclusión de ambos paradigmas, y a que este enfoque

sacrificaba customización en pos de un modelo más rígido, se optó por descartar la idea. De cualquier manera, sería una gran alternativa a ofrecer en una versión más desarrollada.

Otra posibilidad que se consideró (y descartó) en primera instancia es la de agrupar objetos. Esto permitiría posicionarlos relativamente de manera más organizada, ahorrando además el tedio que podría resultar ubicarlos uno por uno. Cualquier persona que haya utilizado aplicaciones como *PowerPoint* o *Canva* puede dar testimonio de la utilidad de la función, pero en este caso se dejó de lado para favorecer otras partes del lenguaje. Esta decisión se tomó sin cargo de consciencia debido a que el lenguaje no cuenta con la complejidad suficiente para hacer imperativa a esta función.

Una extensión posible sería permitir que múltiples objetos con el mismo nombre convivan dentro de una misma diapositiva. Actualmente, si se definiera un texto "*heading1*" con ciertas propiedades de CSS, sólo podría utilizarse una única vez dentro de cada filmina. Esta decisión se tomó en pos de simplificar la implementación del lenguaje, pero se considera altamente deseable modificar esto a futuro (lo cual significaría modificar el identificador interno del compilador para cada variable).

En relación al posicionamiento, como se adelantó en la sección anterior, como primera instancia de mejora para el proyecto se buscaría expandir las capacidades de posicionamiento de objetos para permitir que sea absoluto, o inclusive agregar control sobre el eje z. Esto potenciaría la capacidad de customización de cada diapositiva, con lo cual se considera que agregaría usabilidad y gran valor al usuario.

Adicionalmente, hay varias herramientas que existirían en una versión expandida de *Pressed*, tales como la posibilidad de agregar formas de todo tipo (como flechas, círculos, etc.), gráficos, tablas y videos. Ya fuera del contenido de una filmina, sería idóneo poder agregar notas al pie de cada una. La implementación de estas funciones ampliaría las capacidades del lenguaje de manera no deleznable, acercándolo a plataformas de creación de presentaciones comerciales.

En cuanto a la ampliación de funcionalidades ya presentes, podrían agregarse nuevas animaciones tanto a objetos específicos (entrar desde un lado de la diapositiva, barrido) como entre las filminas (*fade*, avioncito de papel, cubo Rubik) y disponibilizar a nuevas propiedades, mejorando considerablemente el abanico de posibilidades para el usuario y aumentando el atractivo estético de una presentación hecha en *Pressed*.

Por último, en una versión extendida del proyecto se buscaría que -al fallar la compilación- no aparezcan *memory leaks* producto del análisis sintáctico de Bison (dado que el mismo aborta al encontrar un error sin liberar las estructuras y módulos).

Como puede observarse, el proyecto cuenta con la posibilidad de expandirse en un número de direcciones, producto del dominio mismo al que pertenece. En este sentido, se considera que *Pressed* tiene el potencial de ser una herramienta de amplia utilidad para el usuario final, para lo cual se precisa continuar el desarrollo del mismo.

6. Bibliografía

- ***glib.h*** (versión 2.0): Utilizada por su implementación de *HashTables* y arreglos. Ver GLib – 2.0.