Trabajo Práctico Especial
Autómatas, Teoría de Lenguajes y Compiladores

Grupo JIFI

Integrantes:

- Griggio, Juan Gabriel. Legajo: 59092
- Méndez, Ignacio Alberto. Legajo: 59058
- Navarro, Franco. Legajo: 59055
- Villanueva, Ignacio. Legajo: 59000

Tabla de contenidos

Idea y objetivo del lenguaje

Nuestro lenguaje se llama U3D (de “Ultra 3D”). Es un lenguaje de programación de alto nivel que permite dibujar elementos en pantalla de manera rápida y sencilla.

Nos basamos en Processing, un proyecto basado en Java para cumplir con este objetivo. Si bien lo logra, nosotros queríamos hacer algo todavía más fácil y rápido. Es por eso que U3D usa como intermediario a Processing, que luego él mismo termina convirtiéndose en código Java. Por ejemplo: para dibujar alguna figura en pantalla, al usar Processing hay que escribir varias líneas de código, y con U3D eso se simplifica considerablemente.

Nuestro mayor enfoque, y aquello que nos destaca, es la posibilidad de definir **figuras complejas componiendo figuras más simples.** Es decir, uno puede crear una figura “casa” componiendo una figura de tipo “box” y “pyramid”. Esto permite crear mayores niveles de abstracción a partir de simples figuras.

U3D no requiere mucha configuración, ni conocimientos de programación avanzados para poder comenzar a usarlo.

Instructivo de uso

1. Descargar e instalar la JVM (Java Virtual Machine)
2. Descargar e instalar Lex y Yacc con el comando sudo apt-get install bison flex
3. Descargar la última versión de Processing de https://processing.org/download/
4. Clonar el proyecto Git.
5. Setear la variable de entorno **U3DRE_PATH** con el path hacia la carpeta descargada de processing (por ejemplo, la carpeta se puede haber descargado con el nombre “processing-3.5.4”).
    a. Para setear la variable de entorno, en Linux se puede usar el comando: **export U3DRE_PATH=<path>**
6. Estando en la carpeta root del proyecto, compilar el programa compilador con:
    a. **make** compila el proyecto de la manera estándar y genera el ejecutable **u3dc**.
    b. **make -o <carpeta-de-output>** hace lo mismo pero permite elegir el nombre de la carpeta de output.
    c. **make -w** compila el proyecto para el sistema operativo Windows.
    d. **make -l** compila el proyecto para el sistema operativo Linux (**opción por default)**.
    e. **make -p** solamente genera el output de código Processing.
7. Correr el compilador con **./u3dc <path_hacia_algún_test>**. Si el programa test es correcto, se generará una carpeta **U3Dout**. Allí dentro, se encuentra el ejecutable llamado **U3Dout** que al correr llamará a Processing para que muestre visualmente el resultado.

Consideraciones realizadas

El trabajo práctico se desarrolló usando Lex como analizador léxico y Yacc como analizador sintáctico. Se utilizaron los conocimientos aprendidos en las clases sobre estos temas, junto con refuerzos de internet.

El lenguaje de salida generado es Java, ya que se encontró que era la mejor alternativa en conjunto a Processing para poder dibujar figuras con nuestro lenguaje. Por otra parte, todos los miembros del equipo se sentían cómodos y tienen experiencia con Java, por lo que nos ayudó a agilizar el desarrollo.

Desarrollo del Trabajo Práctico

Comenzamos investigando acerca de las distintas opciones para graficar. Pensamos en usar alguna librería de Python para graficar, hicimos pruebas y no nos convencieron. Luego intentamos con Java y OpenGL, pero tampoco nos resultó. Finalmente nos decidimos por Processing.

Luego comenzamos a definir la gramática, crear el árbol de parsing y realizar los parsers correspondientes. Dado el árbol creado, lo que sigue es recorrerlo para asegurarnos de que no hayan errores de sintaxis. Si hay alguno, nos encargamos de atraparlos para que no lleguen a nuestro lenguaje de salida.

Para la parte final del trabajo, nos dividimos en dos sub equipos: el primero se encargó de todo lo que respecta al dibujo de figuras en sí, todo lo que tiene que ver con la traducción a código de Processing y la creación de programas de prueba, y el segundo se encargó de todo el tema de definición y asignación de variables, constantes, expresiones lógicas, expresiones aritméticas, condicionales, ciclos, etc.

Aspectos del lenguaje

Variables

U3D soporta variables de tipo entero "**int**", cadena de caracteres "**string**", números decimales "**float**", vectores "**(1,2,3)**" y variables de tipo figura "**figure"**.

Figuras

U3D permite definir figuras para luego dibujarlas. Las figuras soportadas en nuestro lenguaje son:

- Sphere
- Box
- Pyramid
- Composite

Cada figura se puede crear con atributos. Entre ellos se encuentran: la posición, la rotación, el color, la escala y la figura hija.

Una vez definida alguna de estas figuras, U3D permite crear figuras más complejas. Por ejemplo, es posible crear una figura "casa" usando una "box" como cuerpo de la casa y "pyramid" como su techo. Esto hace a U3D un lenguaje que se destaca entre sus competidores, ya que permite una muy accesible escalabilidad para la creación de figuras mucho más complejas y le da al usuario la posibilidad de crear prácticamente cualquier forma que desee.

Funciones

Nuestro lenguaje incluye algunas funciones que facilitan mucho el dibujo de figuras en pantalla. Lamentablemente no es posible que el usuario defina sus propias funciones, pero esto es algo que se puede implementar en un futuro. Cada función se puede llamar con valores primitivos o con variables, y tienen overload (es decir, algunas funciones se pueden llamar con distintos argumentos).

Las funciones soportadas actualmente son las siguientes:

- drawFigure(figure)
- print(string)
- translateFigure(figure, vector3) / translateFigure(figure, float, float, float)
- rotateFigure(figure, vector3) / rotateFigure(figure, float, float, float)
- scaleFigure(figure, vector3) / scaleFigure(figure, float, float, float)
- addColorFigure(figure, vector3int) / addColorFigure(figure, int, int, int)
- setFigurePosition(figure, vector3) / setFigurePosition(figure, float, float, float)
- setFigureRotation(figure, vector3) / setFigureRotation(figure, float, float, float)
- setFigureScale(figure, vector3) / setFigureScale(figure, float, float, float)
- setFigureColor(figure, vector3int) / setFigureColor(figure, int, int, int)
- setWindowSize(int, int)
    - Setea el tamaño de la ventana en donde se verá todo el dibujo.
- setBackground(vector3int) / setBackground(int, int, int)

- setColorModeToHSB()
- lights(int)
    - Permite ver a las figuras con iluminación de ambiente.

Logs

Proporcionamos un archivo de logging llamado **u3dc.log** que se genera en la carpeta *root* cada vez que se genera la carpeta U3Dout. En este archivo se escriben útiles indicaciones de qué es lo que está sucediendo en la ejecución del compilador.

Además, en u3dc.log se muestra la estructura del árbol de parsing una vez terminado el parseo (en el caso de que el parseo haya sido exitoso).

Si se compila el proyecto con **make u3d_debug**, al archivo de logging se añadirán muchas más indicaciones de nivel DEBUG.

Programas de prueba

Incluímos 5 programas de prueba para demostrar las capacidades de esta primera versión del lenguaje U3D.

Descripción de la Gramática

G = <Vn, Vt, start, P >
Vt = { SETTINGS_BLOCK, DRAW_BLOCK, END_BLOCK, ENDL, EQUAL, COLON, PLUS, MINUS, TIMES, DIVIDE, MODULE, INT_TYPE, FLOAT_TYPE, FIGURE_TYPE, FUNCTION_TYPE, BOOLEAN_TYPE, STRING_TYPE, IDENTIFIER, INTEGER, FLOAT, STRING, BOOLEAN, BRACKET_OPEN, BRACKET_CLOSE, OPEN, CLOSE, COMMA, DOT, WHILE, IF, AND, OR, GT, LT, GE, LE, EQ, NEQ, CONST}

Vn = {start, declaration_list, block_list, block, definition, draw, define_figure, figure_atribute_list, figure_atribute, identifier, value, vector_value, numeric_value, string_value, boolean_value, code_block, code_line, variable_creation, variable_value_update, constant_creation, if, while, conditional, numeric_expression, function_identifier, function_call, parameter_list }

P = {
start → declaration_list block_list | block_list | declaration_list
declaration_list → declaration_list definition | definition
block_list → block_list block | block
block → SETTINGS_BLOCK END_BLOCK | SETTINGS_BLOCK function_call END_BLOCK | draw
definition → define_figure | variable_creation | constant_creation
draw → DRAW_BLOCK END_BLOCK | DRAW_BLOCK code_block END_BLOCK

define_figure → FIGURE_TYPE IDENTIFIER EQUAL BRACKET_OPEN figure_atribute_list BRACKET_CLOSE
figure_atribute_list → figure_atribute_list figure_atribute | figure_atribute
figure_atribute → identifier COLON value ENDL
identifier → IDENTIFIER

value → numeric_value | string_value | boolean_value | vector_value | identifier
vector_value → OPEN numeric_value COMMA numeric_value COMMA numeric_value CLOSE
numeric_value → INTEGER | FLOAT
string_value → STRING
boolean_value → BOOLEAN

code_block → codee_block code_line | code_line
code_line → if | while | variable_creation | variable_value_update | function_call | constant_creation

variable_creation → INT_TYPE identifier EQUAL numeric_expression ENDL | STRING_TYPE identifier EQUAL string_value ENDL | FLOAT_TYPE identifier EQUAL numeric_expression ENDL | INT_TYPE identifier EQUAL identifier ENDL | STRING_TYPE identifier EQUAL identifier ENDL | FLOAT_TYPE identifier EQUAL identifier ENDL | INT_TYPE identifier ENDL | STRING_TYPE identifier ENDL | FLOAT_TYPE identifier ENDL | BOOLEAN_TYPE identifier EQUAL BOOLEAN ENDL | BOOLEAN_TYPE identifier ENDL

variable_value_update → identifier EQUAL numeric_expression ENDL | identifier EQUAL string_value ENDL | identifier EQUAL boolean_value ENDL | identifier EQUAL identifier ENDL

constant_creation → CONST INT_TYPE identifier EQUAL numeric_expression ENDL | CONST STRING_TYPE identifier EQUAL string_value ENDL | CONST FLOAT_TYPE identifier EQUAL numeric_expression ENDL | CONST BOOLEAN_TYPE identifier EQUAL boolean_value ENDL | CONST INT_TYPE identifier EQUAL identifier ENDL | CONST STRING_TYPE identifier EQUAL identifier ENDL | CONST FLOAT_TYPE identifier EQUAL identifier ENDL | CONST BOOLEAN_TYPE identifier EQUAL identifier ENDL

if → IF OPEN conditional CLOSE BRACKET_OPEN code_block BRACKET_CLOSE

while → WHILE OPEN conditional CLOSE BRACKET_OPEN code_block BRACKET_CLOSE

conditional → conditional AND conditional | conditional OR conditional | numeric_expression LT numeric_expression | numeric_expression GT numeric_expression | numeric_expression LE numeric_expression | numeric_expression GE numeric_expression | numeric_expression EQ numeric_expression | numeric_expression NEQ numeric_expression | string_value EQ string_value | string_value NEQ string_value | boolean_value EQ boolean_value | boolean_value NEQ boolean_value | identifier GT numeric_expression | identifier LE numeric_expression | identifier GE numeric_expression | identifier EQ numeric_expression | identifier NEQ numeric_expression | identifier EQ string_value | identifier NEQ string_value | identifier EQ boolean_value | identifier NEQ boolean_value | numeric_expression LT identifier | numeric_expression GT identifier | numeric_expression LE identifier | numeric_expression GE identifier | numeric_expression EQ identifier | numeric_expression NEQ identifier | string_value EQ identifier | string_value NEQ identifier | boolean_value EQ identifier | boolean_value NEQ identifier | identifier LT identifier | identifier GT identifier | identifier LE identifier | identifier GE identifier | identifier EQ identifier | identifier NEQ identifier

numeric_expression → numeric_expression PLUS numeric_expression | numeric_expression MINUS numeric_expression | numeric_expression TIMES numeric_expression | numeric_expression DIVIDE numeric_expression | numeric_expression MODULE numeric_expression | identifier PLUS numeric_expression | identifier MINUS numeric_expression | identifier TIMES numeric_expression | identifier DIVIDE numeric_expression | identifier MODULE numeric_expression | numeric_expression PLUS identifier | numeric_expression MINUS identifier | numeric_expression TIMES identifier | numeric_expression DIVIDE identifier | numeric_expression MODULE identifier | identifier PLUS identifier | identifier MINUS identifier | identifier TIMES identifier | identifier DIVIDE identifier | identifier MODULE identifier | numeric_value

function_identifier → IDENTIFIER

function_call → function_identifier OPEN CLOSE ENDL | function_identifier OPEN parameters_list CLOSE ENDL

parameters_list → parameters_list COMMA value | value
}

Dificultades encontradas

1. La primera dificultad encontrada fue encontrar un lenguaje de salida que pueda ser convertido a gráficos. Habíamos investigado sobre OpenGL: nuestra idea era que nuestro lenguaje genere código Java que use OpenGL y dibuje figuras. Sin embargo, se nos complicó mucho configurarlo para poder hacer que el usuario no tenga que instalarse muchas cosas para poder mínimamente correr nuestro proyecto.
   En ese momento, encontramos Processing, que fue muy simple de usar y de configurar, por lo que decidimos usarlo en el trabajo práctico.
2. Otra dificultad que se tuvo al comienzo del trabajo práctico fue la aparición de los clásicos conflictos Shift/Reduce. Tuvimos que detenernos un tiempo a repensar las producciones hasta que finalmente pudimos eliminar todos los conflictos.
3. Una dificultad más fue el hacer andar las variables de tipo booleanas. Si bien pareciera que el comportamiento debería ser similar a las variables de tipo enteras, decimales o cadenas de caracteres, tuvimos muchos problemas a la hora de que funcionen con nuestra gramática.

Posibles extensiones

1. La mayor posible extensión que tiene U3D es incrementar la cantidad de figuras soportadas. Ya que nuestro lenguaje construye una figura a partir de otra, complejizar las formas dibujadas es relativamente sencillo de implementar. De esta forma, creemos que podemos agregar figuras mucho más complejas al lenguaje.
2. Otra posible extensión es el hecho de generar conjuntos de elementos. Por ejemplo, con un simple llamado a función poder dibujar muchas casas, incluso con leves diferencias. Esto creemos que no es tan fácil de implementar, pero lo vemos como una buena idea a futuro.
3. Por otro lado, hay muchas opciones de sintaxis que por el momento no son soportadas y se pueden agregar a U3D. Por ejemplo: incrementar o decrementar variables haciendo i++, i+= x, etc.
4. Otra extensión muy importante y necesaria es el soporte a variables de tipo booleanas. Como fue discutido en las dificultades encontradas, tuvimos problemas al implementar este tipo de dato y finalmente no es soportado por nuestro lenguaje.
5. Para el siguiente ejemplo, nuestro lenguaje no logra atrapar el error antes de que llegue al lenguaje de salida: **float a = 5.5; int b = a + 2;** Esto se debe a que no se logró reconocer que la expresión a la derecha del "=" va a terminar siendo un número de tipo "float"**.** Sí se logra atrapar este error si la parte derecha es un "float" solamente. Este fallo se debe a que por el momento las expresiones aritméticas no guardan información acerca de su tipo de retorno.
6. Concatenación de strings: por el momento, no se puede hacer.
7. Mejor manejo de luces, cámaras y texturas.
8. Definición de funciones propias.
9. Más funciones en su "librería estándar", como por ejemplo una función que detecte cuando 2 figuras colisionan.
10. Se podría desarrollar una forma de obtener el input del usuario, para así generar valores en base a lo que ingrese.