V# : Un lenguaje para la creación de medios audivisuales

Ciencia de la Computación — Compiladores - CCOMP8-1

José Manuel Cornejo Lupa
Guido Luis Tapia Oré
José Eduardo Zenteno Monteagudo

3 de enero de 2021

Índice

1. Introducción

El lenguaje a ser elaborado durante el presente curso se oriento a la edición de vídeos. Siendo que para esta tarea ya existen múltiples software, se tomara inspiración de estos para la elaboración del lenguaje. Asimismo, se opto por utilizar palabras clave provenientes del ingles, debido a la facilidad gramatical que presenta este idioma con respecto a otros y a la diversificación de este idioma en diversos lenguajes de programación ya establecidos. El presente documento pretende dar a conocer los conceptos básicos y/o esenciales del lenguaje creado en cuestión.

2. Manual de referencia

La elaboración de este fue presentada en la segunda revisión y cuenta con una versión actualizada, la cual se anexa al final de este documento y puede ser accedido mediante el siguiente link:

`https://drive.google.com/file/d/1N1saRRNc4LqN0XE_7_v-2pzmRrrd9xWj/view?usp=sharing`

Este documento fue elaborado tomando como referencia los ejemplos presentados de los manuales de C++ y Python.

3. Descripción del desarrollo del proyecto

3.1. Herramientas y lenguajes usados

El proyecto fue desarrollado en su totalidad usando el lenguaje python3, y se busco realizar un interpreter para python3 usando la librería moviepy como lenguaje intermedio.

3.2. Componentes principales del traductor

La elaboración y los componentes se vieron fuertemente inspirados por las actividades realizadas durante las horas prácticas de curso, se contemplan tres elementos esenciales para la traducción, y un elemento adicional para la recuperación de errores; los cuales se describen a continuación.

3.2.1. Analizador Lexico

El análisis léxico se da gracias a un tokenizador, el cual analiza carácter por carácter el código ingresado en texto plano; y da como salida una lista de tokens ordenados, considerando la siguiente estructura de clase para estos:

```
class Token:
    palabra = ""
    indice = (-1,-1)
    tipo = ''
```

Los atributos dentro de la clase Tokens obedece a las siguientes consideraciones:

- **palabra:** Almacena una copia del conjunto de caracteres identificados como un elemento dentro del texto plano, a fin de su uso en la etapa de traducción.
- **indice:** Almacena el numero de linea y posición en esta del primer carácter del elemento identificado, su presencia se debe a su gran aporte en la recuperación de errores.
- **tipo:** Almacena el tipo del token que se reconoció, los tipos considerados son los siguientes: error, int, float, variable, string, time, comma, equal, endl, iniScope, endScope.

Asimismo, el tokenizador trabaja en conjunto con un diccionario de operadores y un pool de palabras reservadas los cuales presentan la siguiente definición:

```
operadores={',':"comma",'=':"equal",';':"endl",'{':"iniScope",\
            '}':"endScope"}
reservadas={"Text","Resources","Render","Timeline","Video","Visual",\
            "Audio","vid","aud","img","color","size","font","name",\
            "extension","directory","import","as","insert","from",\
            "to","on","at","for","px","Properties"}
```

El algoritmo del analizador se describe a continuación:

```
Datos de Entrada:
Texto plano txt con el codigo ingresado por el usuario
Algoritmo:
Para cada linea en txt:
    Para cada palabra en linea:
        Si el primer caracter de palabra es el caracter comilla('')
            Si palabra finaliza con el caracter comilla('') y
            no presenta el caracter slash(\)
                Anhadir Token de string
            Sino
                Anhadir Token de error
        Sino si el primer caracter de palabra es un numero
            Si palabra solo presenta numeros
                Anhadir Token de int
            Sino si palabra tiene el formato para el tiempo
                Anhadir Token de time
            Sino si palabra presenta numeros y un caracter punto(.)
                Anhadir Token de float
            Sino
                Anhadir Token de error
        Sino si el primer caracter de palabra es una letra o
        el caracter guion(-) o subguion(_)
            Si palabra es una palabra reservada
                Anhadir Token de keyWord
            Sino si palabra solo contiene caracteres alphanumericos,
            guiones(-) y subguiones(_)
                Anhadir Token de variable
            Sino
                Anhadir Token de error
        Sino si el primer caracter de palabra es un operador
            Anhadir Token de operador
        Sino
            Anhadir Token de error
```

3.2.2. Analizador Sintáctico

Este se elaboro considerando el uso de una gramática LL(1), la cual se encuentra detallada en el manual de referencia. Este presenta dos parametros de ingreso: una gramática preprocesada y el conjunto de tokens obtenidos gracias al analizador léxico. Se deben considerar tres clases que serán de gran importancia para el análisis sintáctico, estas se presentan a continuación:

```
class Produccion:
    izq=""
    der=[]

class TAS:
    terminales=[]
    noterminales=[]
    tablaSintactica={}

class Gramatica:
    produccion = []
    terminales = set()
    noterminales = set()
    inicial = ""
    primeros={}
    siguientes = {}
    tas=None
```

La clase Produccion es una posibilidad de derivación para un nodo no terminal, sus atributos corresponden a lo siguiente:

- **izq:** Nombre de nodo no terminal.
- **der:** Derivacion expresada en una lista ordenada de strings que corresponden a nombres de nodos terminales y no terminales.

La clase TAS es una tabla sintáctica usada a forma de predicción para orientar la forma en la cual se realizaran las derivaciones en el análisis sintáctico, sus atributos corresponden a lo siguiente:

- **terminales:** Lista de nodos terminales obtenidos a partir de una gramática.
- **noterminales:** Lista de nodos no terminales obtenidos a partir de una gramática.
- **tablaSintactica:** Diccionario Bidimensional que almacena predicciones de derivación a partir dos keys: un nodo no terminal, y un nodo terminal; debido a que se considero una gramática sin backtracking, solo se almacena una única posibilidad de derivación.

Y, como la clase mas importante, tenemos a Gramatica, la cual servirá para almacenar una gramática procesada considerando atributos importantes en esta:

- **produccion:** Lista de clases Produccion.
- **terminales:** Pool de strings representando a los nodos terminales.
- **noterminales:** Pool de strings representando a los nodos terminales.
- **inicial:** Nombre del nodo que actuara como raíz al generar el árbol sintáctico.
- **primeros:** Diccionario de primeros nodos según nodo no terminal.
- **siguientes:** Diccionario de predicciones de nodos siguientes según nodo no terminal.
- **tas:** tabla sintáctica construida a partir de la propia gramática.

Todos los elementos vistos con anterioridad actuaran en conjunto en el algoritmo de análisis, obteniendo como resultado un árbol sintáctico del cual su estructura sera revisada en el siguiente punto.

3.2.3. Traductor

Este se da gracias a la función interpret que cada nodo del árbol sintáctico posee, se considera como entrada al árbol sintáctico y la salida correspondería a un código intermedio en el lenguaje python. Para entender la traducción resulta necesario revisar las dos clases abstractas que actuarán como nodos en los arboles sintácticos.

```
class NNTerminal:
        prod = {}
        etiqueta = ''
        padre = None
        siguiente = None

class NTerminal:
        val = ''
        etiqueta = ''
        padre = None
        siguiente = None
```

En base a estas dos clases abstractas se elaboraran las demás usando una clase constructor, la cual estará encargada de generar archivos.py a partir de reglas definidas en el formato de texto plano. De esta forma se elimina la dependencia de múltiples archivos, por lo que la posibilidad de perder segmentos de código se ve mitigada. Debido a la gramática elegida, se cuenta con 78 nodos especializaciones, todos estos contendrán una función interpret diferente, una de esas se detalla a continuación:

```
from NNTerminal import NNTerminal
class NodoINSAUDIO(NNTerminal):
  def __init__(self,padre):
    super().__init__('INSAUDIO',padre)
    self.prod['insert1'] = None
    self.prod['variable1'] = None
    self.prod['from1'] = None
    self.prod['time1'] = None
    self.prod['to1'] = None
    self.prod['time2'] = None
    self.prod['on1'] = None
    self.prod['time3'] = None
  def interpret(self):
    return '_timeline_aud_.append('+self.prod['variable1'].interpret()+\
           '.coreader().subclip('+self.prod['time1'].interpret()+','+\
           self.prod['time2'].interpret()+').set_start(t='+\
           self.prod['time3'].interpret()+'))'
```

3.2.4. ErrorLog

Para la recuperación de errores se uso la estructura propuesta en clase, la idea de esta era almacenar los errores encontrados durante la ejecución como nuevas instancias, a fin de ayudar al programador en su labor.

```
DiccionarioError={1:'Error_lexico:_No_se_encontro_comilla_de_cierre_en_string',\
                  2:'Error_lexico:_Uso_no_permitido_de_caracter_(\\)',\
                  3:'Error_lexico:_Error_de_escritura_en_valor_numerico',\
                  4:'Error_lexico:_Error_de_escritura_en_nombre_de_variable',\
                  5:'Error_lexico:_Operador_o_caracter_no_reconocido',\
                  6:'Error_sintactico:_Falta_operador_llave_({)_que_indique_inicio_del_Scope',\
                  7:'Error_sintactico:_Falta_operador_llave_(})_que_indique_final_del_Scope',\
                  8:'Error_sintactico:_Falta_operador_punto_y_coma_(;)_que_indique_fin_de_linea'}
class Log:
    lista_errores=[]
    lista_warnigs=[]
```

La clase Log tiene dos atributos lista, los cuales servirán para almacenar los errores o warning encontrados en los diferentes análisis, se consideraron 2 categorías de errores: los errores léxicos, los cuales son identificados por el tokenizador, y se refieren a un carácter no reconocido o a errores en la escritura de los diferentes elementos; los errores sintácticos se refieren a elementos delimitantes faltantes, como serian los puntos y coma o las llaves. En el caso de los primeros, el encontrar un error no significara interrumpir el análisis, en cambio los errores sintácticos si lo harán, al hacer entrar al analizador en modo pánico.

3.3. Interfaces entre componentes

Al buscar cierta independencia entre los diferentes componentes del interpreter para evitar errores, la interaccion entre estos se centró en una función principal main, la cual encapsulaba toda la interacción a fin de lograr el esquema propuesto, a continuacion se presenta el fragmento de codigo con la funcion main.

```
def main():
#Cargar y procesar la grm tica
    gramaticaEditor=Gramatica()
    gramaticaEditor.cargar(open("gramatica.txt","r"))
    tabla=gramaticaEditor.crearTabla()
    errorLog.inicializar()
#Creacion de interpreter a partir de la gramtica
    const=Constructor(gramaticaEditor)
#Analisis Lexico
    tokens=analizadorLexico(open("codigoInputPrueba/valido1.txt","r"),errorLog)
#Analisis Sintactico
    arbolRaiz=analizadorSintactico(gramaticaEditor, tokens,errorLog)
#Recuperacion de errores
    if (arbolRaiz!=None and errorLog.print()):
#Generacion de codigo intermedio
        f = open("output.py", "w")
        f.write(arbolRaiz.interpret())
        f.close()
        print("--------- codigo analizado con exito ---------")
#Ejecucion de codigo intermedio
        exec(open("output.py").read())
    else:
        print("--------- Error Semantico en code1 ---------")
    errorLog.print()
```

3.4. Testing y ejemplos

3.4.1. Ejemplos Válidos

1. **Ejemplo 1 :** En este primer ejemplo se usará una definición de estilo de texto, dos recursos importados, y tres tracks del timeline.

```
Text titulo1{
  color = "white";
  font = "calibri";
  size = 15;
}

Render Proyecto1{
  Properties{
    extension= "mp4";
    directory= "D:/UCSP_2020/2020-2/Compiladores/proyecto_final/entrega_Final/prueba1/";
  }
  Resources{
    import vid "video.wmv" as recurso1;
    import aud "music.mp3" as recursoMusical2;
  }
  Timeline{
    Video{
      insert recurso1 from 00:01 to 00:10 on 00:20;
    }
    Visual{
      insert "Prueba_de_video" as titulo1 at 0,0 px on 00:00 for 00:05;
    }
    Audio{
      insert recursoMusical2 from 00:01 to 00:10 on 00:10;
    }
  }
}
```

El código intermedio resultante corresponde a:

```
from moviepy.editor import *
_codecs_={'mp4':'mpeg4','avi':'rawvideo','ogv':'libvorbis','webm':'libvpx'}
_W_ = 600
_H_ = 400
_name_='default'
_extension_= 'mp4'
_directory_= 'C:/Users/LEGION/Desktop/'
_timeline_vid_=[]
_timeline_aud_=[]

#Definicion Estilos

titulo1 = ['white','calibri',10]
titulo1[0]="white"
titulo1[1]="calibri"
titulo1[2]=15

#Render

#Render Propiedades

_extension_ = "mp4"
_directory_ = "D:/UCSP_2020/2020-2/Compiladores/proyecto_final/entrega_Final/prueba1/"

#Render Recursos

recurso1 = VideoFileClip(_directory_+"video.wmv")
_Waux_,_Haux_ = recurso1.size
_W_= max(_W_,_Waux_)
_H_=max(_H_,_Haux_)
recursoMusical2 = AudioFileClip(_directory_+"music.mp3")

#Render Timeline

#Tracks de Videos

_timeline_vid_.append(recurso1.subclip(1,10).set_start(t=20))
_timeline_aud_.append(_timeline_vid_[-1].audio)

#Tracks de Recursos Visuales

_timeline_vid_.append(TextClip("Prueba_de_video",color=titulo1[0],\
font=titulo1[1],fontsize=titulo1[2]).set_duration(5).set_start(t=0).set_position((0,0)))

#Tracks de Audio

_timeline_aud_.append(recursoMusical2.coreader().subclip(1,10).set_start(t=10))

_cc_vid_ = CompositeVideoClip(_timeline_vid_,size = (_W_,_H_))
_cc_vid_.audio = CompositeAudioClip(_timeline_aud_)
_cc_vid_.write_videofile(_directory_+_name_+'.'+_extension_,fps=30, codec=_codecs_[_extension_])
```

2. **Ejemplo 2 :** En este ejemplo se ignora la definicion de estilos de texto, en cambio se importan todos los tipos de recursos posibles (3), y se usa tres tracks del timeline.

```
Render Proyecto1{
  Properties{
    extension = "mp4";
    directory = "D:/UCSP_2020/2020-2/Compiladores/proyecto_final/entrega_Final/prueba2/";
  }
  Resources{
    import vid "video.mp4" as recurso1;
    import aud "music.mp3" as recursoMusical2;
    import img "fondo.jpg" as imagen1;
  }
  Timeline{
    Video{
      insert recurso1 from 00:01 to 00:10 on 00:20;
    }
    Visual{
      insert imagen1 on 00:00 for 00:20;
    }
    Audio{
      insert recursoMusical2 from 00:01 to 00:21 on 00:00;
    }
  }
}
```

El código intermedio resultante corresponde a:

```
from moviepy.editor import *
_codecs_={'mp4':'mpeg4','avi':'rawvideo','ogv':'libvorbis','webm':'libvpx'}
_W_ = 600
_H_ = 400
_name_='default'
_extension_= 'mp4'
_directory_= 'C:/Users/LEGION/Desktop/'
_timeline_vid_=[]
_timeline_aud_=[]

#Render

#Render Propiedades

_extension_ = "mp4"
_directory_ = "D:/UCSP_2020/2020-2/Compiladores/proyecto_final/entrega_Final/prueba2/"

#Render Recursos

recurso1 = VideoFileClip(_directory_+"video.mp4")
_Waux_,_Haux_ = recurso1.size
_W_= max(_W_,_Waux_)
_H_=max(_H_,_Haux_)
recursoMusical2 = AudioFileClip(_directory_+"music.mp3")
imagen1=_directory_+"fondo.jpg"

#Render Timeline

#Tracks de Videos

_timeline_vid_.append(recurso1.subclip(1,10).set_start(t=20))
_timeline_aud_.append(_timeline_vid_[-1].audio)

#Tracks de Recursos Visuales

_timeline_vid_.append(ImageClip(imagen1, duration= 20 ).set_start(t=0))

#Tracks de Audio

_timeline_aud_.append(recursoMusical2.coreader().subclip(1,21).set_start(t=0))

_cc_vid_ = CompositeVideoClip(_timeline_vid_,size = (_W_,_H_))
_cc_vid_.audio = CompositeAudioClip(_timeline_aud_)
_cc_vid_.write_videofile(_directory_+_name_+'.'+_extension_,fps=30, codec=_codecs_[_extension_])
```

3.4.2. Ejemplos Erróneos

1. **Ejemplo 1 :** En este primer ejemplo de errores se visualiza que un scope no esta debidamente delimitado, dado que no cuenta con la llave de cierre.

```
Text titulo1{
        color = "white";
        font = "calibri";
        size = 15;

Render Proyecto1{
        Properties{
                extension= "mp4";
                directory= "C:/Users/LEGION/Desktop/";
        }
        Resources{
                import vid "video.wmv" as recurso1;
                import aud "music.mp3" as recursoMusical2;
        }
        Timeline{
                Video{
                        insert recurso1 from 00:01 to 00:10 on 00:20;
                }
                Visual{
                        insert "Prueba_de_video" as titulo1 at 0,0 px on 00:00 for 00:05;
                }
                Audio{
                        insert recursoMusical2 from 00:01 to 00:10 on 00:10;
                }
        }
}
```

Debido a la presencia del error, no se generara un codigo intermedio, en cambio obtendremos la siguiente salida:

```
Modo Panico Activado: Error de Scope
------ Error Semantico en code1 ---------
Error 7 en la linea 6, posicion 0
Token:Render
Error sintactico: Falta operador llave (}) que indique final del Scope
```

2. **Ejemplo 2 :** En este ejemplo se cometen múltiples errores sintácticos a fin de demostrar la capacidad de recuperacion de errores del lenguaje presentado.

```
Text titulo1{
        color = "white";
        font = "cali\bri"
        size = 15f;
}
Render Proyecto1{
        Properties{
                extension= "mp4";
                directory= "C:/Users/LEGION/Desktop/";
        }
        Resources{
                import vid "video.wmv" as recurso1;
                import aud "music.mp3" as recursoMusical2;
        }
        Timeline{
                Video{
                        insert recurso1 from 00f:01 to 00:10 on 00:20;
                }
                Visual{
                        insert "Prueba_de_video" as titulo1 at 0,0f px on 00:00 for 00:05;
                }
                Audio{
                        insert recursoMusical2 from 00:01 to 00:10 on 00:10;
                }
        }
}
```

Debido a la presencia del error, no se generara un codigo intermedio, en cambio obtendremos la siguiente salida:

```
Error 2 en la linea 3, posicion 8
Token:"cali\bri"
Error lexico: Uso no permitido de caracter (\)
Error 4 en la linea 3, posicion 9
Token:cali\bri"
Error_lexico:_Error_de_escritura_en_nombre_de_variable
Error_3_en_la_linea_4,_posicion_8
Token:15f
Error_lexico:_Error_de_escritura_en_valor_numerico
Error_3_en_la_linea_20,_posicion_44
Token:0f
Error_lexico:_Error_de_escritura_en_valor_numerico
Error_8_en_la_linea_3,_posicion_9
Token:cali\bri"
Error sintactico: Falta operador punto y coma (;) que indique fin de linea
```

4. Lecciones aprendidas

El presente trabajo ha dejado muchas enseñanzas en todos los miembros del equipo, tanto con respecto a la elaboración de interpretadores como en el desarrollo de software:

- El realizar múltiples análisis no solo obedece a un criterio de orden, sino también de lógica, siendo que este permite realizar una mejor retroalimentación a un usuario.
- El diseño de la gramática es fundamental en el diseño de compiladores, debido a que esta definirá no solo las restricciones que el usuario pueda tener, sino también influirá fuertemente en la carga de trabajo de su implementación.
- Existen múltiples enfoques que pueden ser abordados para la elaboración de un compilador en diferentes etapas.
- La recuperación de errores resulta ser una de las tareas más criticas, debido a la dificultad de su implementación a diferentes niveles y debido a que esta resulta necesaria para la retroalimentación del usuario.
- El diseño de software siempre debe estar ligado desde su concepción a los conocimientos de temas afines, debido a que estos suelen tomar consideraciones similares.

5. Implementación

La implementación del presente trabajo es accesible mediante el siguiente repositorio virtual:

`https://github.com/ginalucia/Compiladores-4/tree/main/entrega%20Final`

No obstante debido a las limitaciones que presenta github, también se consideró realizar una copia con los recursos usados en google drive:

`https://drive.google.com/file/d/1GaeHO1Z_Ghy2Uek2RTkZHPQ-tBXaCSOn/view?usp=sharing`