Etapa 2: Análisis Sintáctico

Benjamin Amos, Douglas Torres

Enero - Marzo 2016

1 Detalles de Implementacion

Para la implementación de esta primera etapa del desarrollo de un interprete del lenguaje de programación 'BOT', se utilizó el lenguaje de *scripting* orientado a objetos *Python*, el cual cuenta con una herramienta generadora de analizadores lexicográficos y sintácticos llamada *PLY*.

La implementación del analizador lexicográfico fue dividida en tres partes. Una para el manejo de la herramienta (el '*lexer*'), otra para el manejo de la estructura de datos, y una última para el programa principal. Esto son tres archivos, **lexBot.py**, **classes.py** y **main.py** respectivamente.

1.1 lexBot.py

En este archivo se encuentran las reglas lexicográficas respecto a los tokens que se admiten en el lenguaje '*BOT*'. Al inicio se observa una declaración global de un objeto de tipo *TokenList*, el cual es una estructura de datos que será explicada en la sección del archivo **classes.py**. Se encuentra aquí para registrar el manejo de errores. A su vez se encuentra la declaración global de la variable *content* que contiene el archivo de entrada que será analizado por el lexer en un string.

1.2 classes.py

Este archivo contiene las estructuras de datos utilizadas para esta etapa, las cuales son **Token** y **TokenList**.

1.2.1 Token

La clase **Token** define un objeto que corresponde a cada token reconocido por el lenguaje y cuyos atributos son un nombre, fila y columna en las que se encuentra el token en el código recibido por la entrada, y un argumento en el caso de que el token sea de tipo numérico, caracter o identificador (*name*, *row*, *column* y *arg* respectivamente).

1.2.2 TokenList

Esta clase contiene dos listas, *TokList* y *ErrList*, una para almacenar todos los tokens encontrados y otra para almacenar errores. Se implementaron métodos sobre esta clase para añadir tokens o errores a sus listas respectivas (*addTok* y *addErr*) y para imprimir los tokens o errores de cada lista (*printTok* y *printErr*). Por último, un método para verificar si la lista de errores está vacía (*notErr*). Esto indica que, si la lista de errores está vacía, la ejecución arrojará una salida correcta.

1.3 main.py

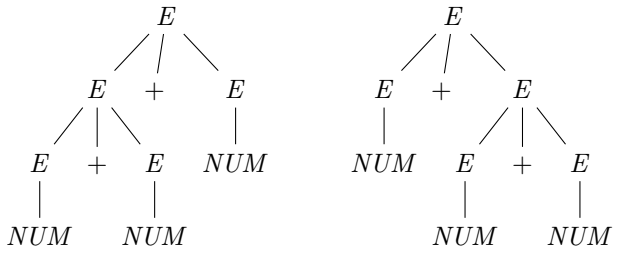
Programa principal en el que ocurre la ejecución del análisis lexicográfico. Se recibe un archivo de entrada y se almacena su contenido para el análisis. La ejecución consiste en dos recorridos al contenido del archivo, una para la detección de errores y, si no se encontró error alguno, se recorre nuevamente para detectar los tokens, almacenarlos e imprimirlos.

2 Sección Teórico-Práctica

En esta sección se presenta el desarrollo y respuestas para las preguntas propuestas para esta etapa.

1. Basados en la gramática *G1* dada:

   (a) Para demostrar que la frase $NUM{+}NUM{+}NUM$ es ambigua, mostraremos que existen dos árboles de derivación distintos que representan a la frase:

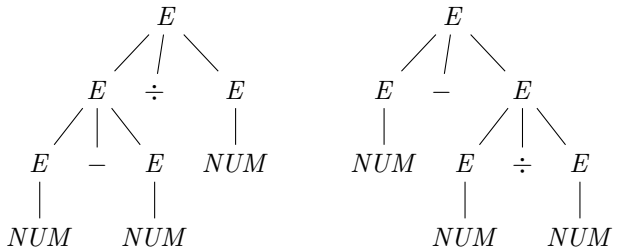


   Notemos entonces que la frase en cuestión es ambigua.

   (b) Sean $Izq(G1)$ y $Der(G1)$ definidas de la siguiente manera:

   - $Izq(G1)$: $Expr \rightarrow Expr + Expr' \mid Expr'$
     $Expr' \rightarrow NUM$
   - $Der(G1)$: $Expr \rightarrow Expr' + Expr \mid Expr'$
     $Expr' \rightarrow NUM$

   (c) En efecto, si importa la forma en la que se asocian las expresiones en esta gramática. Veamos con un ejemplo por qué es relevante. Supongamos que el alfabeto acepta los operadores $-$ y $\div$. Sea la expresión $NUM{-}NUM{\div}NUM$. Construyamos los árboles de derivación respectivos a $Izq(G1)$ y $Der(G1)$ respectivamente:



Notemos que $Izq(G1)$ generaría a la expresión $(NUM-NUM) \div NUM$, mientras que $Der(G1)$ genera a $NUM - (NUM \div NUM)$

2. Basados en la gramática *G2*, tenemos que:

   (a) Se entiende que la gramática *G2* tiene los mismos problemas de ambigüedad que *G1* ya que, $Expr \equiv Instr$, $+ \equiv\ ;$ y $NUM \equiv IS$. Luego, existen dos o más arboles de derivación diferentes para una frase que reconoce la gramática. Las únicas frases no ambiguas de la gramática *G2* son la frase *IS* y la frase *IS* ; *IS* ya que sólo hay un árbol para construirlas.

   (b) Dado que la gramática en este caso reconoce secuencias de instrucciones y no operaciones de expresiones, se puede afirmar que no importa el sentido en el que se asocien las expresiones. Para esto, supongamos que existen dos gramáticas $Izq(G2)$ y $Der(G2)$ tales que:

   - $Izq(G2)$: $Instr \rightarrow Instr\ ;\ Inst'$
     $\quad\quad | \ Instr'$
     $Instr' \rightarrow IS$

   - $Der(G2)$: $Instr \rightarrow Instr'\ ;\ Instr$
     $\quad\quad | \ Expr'$
     $Instr' \rightarrow IS$

   Dado que los árboles de derivación que se presentan son lo mismos que los de *G1*, notemos que las expresiones que se generarían son $(IS; IS); IS$ y $IS; (IS; IS)$. Siendo éstas instrucciones, no importa el orden en el que se realicen.

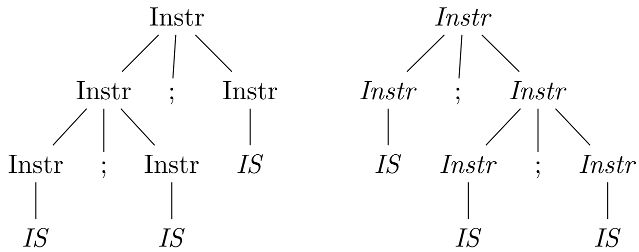
   (c) Veamos una derivación más a la izquierda para la frase $IS; IS; IS$:

   $$Instr \Rightarrow Instr\ ;\ Instr \Rightarrow Instr\ ;\ Instr\ ;\ Instr \Rightarrow Instr\ ;\ Instr\ ;\ IS \Rightarrow Instr\ ;\ IS\ ;\ IS \Rightarrow IS\ ;\ IS\ ;\ IS$$

   Veamos una derivación más a la derecha para la misma frase:

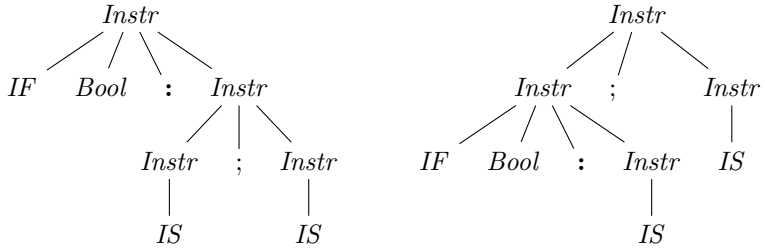
   $$Instr \Rightarrow Instr\ ;\ Instr \Rightarrow Instr\ ;\ Instr\ ;\ Instr \Rightarrow IS\ ;\ Instr\ ;\ Instr \Rightarrow IS\ ;\ IS\ ;\ Instr \Rightarrow IS\ ;\ IS\ ;\ IS$$

   Para ilustrarlo, se muestran dos árboles de derivación diferentes que generan la misma frase:



3. Basados en la gramatica *G3*:

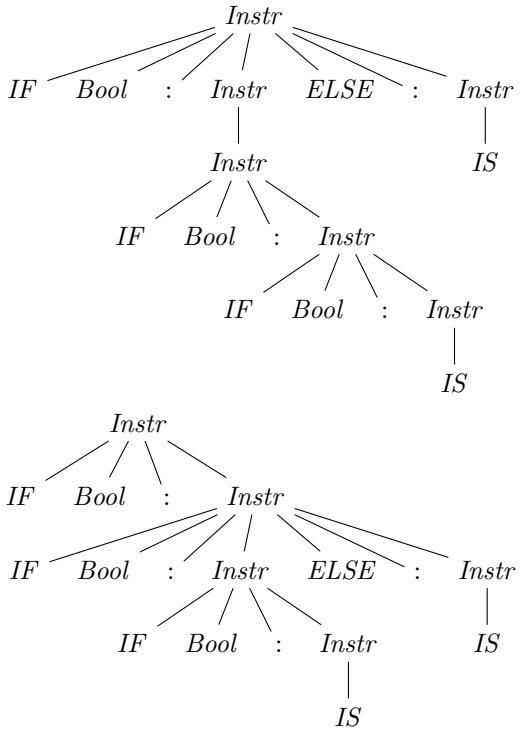
   (a) Mostraremos mediante dos árboles de derivación que la frase *IF Bool* : *IS* ; *IS* es ambigua:



   (b) Una frase *g* de *G3* sin ocurrencias de ; que sea ambigua puede ser:

$$IF\ Bool : IF\ Bool : IF\ Bool : IS\ ELSE : IS$$

   (c) Para mostrar que es ambigua, mostraremos dos árboles de derivación distintos:



   (d) Utilizando llaves, podríamos escribir:

   - Dos interpretaciones de la frase *f* con { y }:
     - *IF Bool* { *IS* ; *IS* }
     - *IF Bool* { *IS* } ; *IS*

- Dos interpretaciones de la frase g con { y }:
  - *IF Bool* { *IF Bool* { *IF Bool* { *IS* } } } *ELSE* { *IS* }
  - *IF Bool* { *IF Bool* { *IF Bool* { *IS* } *ELSE* { *IS* } } }

(e) Utilizando el terminador *end*, podemos escribir:

- Dos interpretaciones de la frase f con { y }:
  - *IF Bool* : *IS* ; *IS end*
  - *IF Bool* : *IS end* ; *IS*
- Dos interpretaciones de la frase g con { y }:
  - *IF Bool* : *IF Bool* : *IF Bool* : *IS end end end ELSE* : *IS end*
  - *IF Bool* : *IF Bool* : *IF Bool* : *IS end ELSE* : *IS end end end*