Interprete de BOT
Etapa 3
Analizador de Contexto

Meggie Sánchez 11-10939, Jorge Marcano 11-10566

07/03/2016

1. Introducción

En el presente proyecto, se nos ha pedido crear un intérprete para el lenguaje llamado "BOT". El desarrollo para el intérprete se dará en 4 etapas: (I) Análisis lexicográfico, (II) Análisis sintáctico y construcción del árbol sintáctico abstracto, (III) Análisis de contexto e (IV) Intérprete final del lenguaje.

Para esta tercera etapa, se ha dado uso de la implementación del analizador de contexto para el lenguaje BOT. Esta etapa incluye la creación de una tabla de símbolos para guardar las distintas variables declaradas en el programa, y mecanismos para la detección de errores de tipo, de redeclaraciones y otros.

Ya dicho anteriormente, ha sido realizado en el lenguaje de programación Python (versión 3.4), con la herramienta ofrecida por este mismo llamada PLY.

2. Implementación

Para la implementación del analizador de contexto, se utilizó una estructura llamada Tabla de Símbolos, la cual se utiliza para guardar la información de los robots declarados para uso en el programa. Esta tabla fue creada como una clase SimTab, la cual contiene un diccionario (la tabla de símbolos en sí) y un .ªpuntador.ª su tabla de símbolos padre para manejar los distintos alcances del programa. El diccionario utilizado funciona de la siguiente manera: Cada vez que ocurre una declaración en el programa, dicha declaración es colocada dentro de la tabla de símbolos, usando su identificador (nombre del robot) como clave, y como valor, una tupla de la forma (tipo, comportamientos) en donde tipo es el tipo del robot declarado, y comportamientos es otro diccionario en el cual se incluyen los comportamientos de dicho robot en las situaciones de .ªctivación", "desactivaciónz "default", de forma que la clave en ese diccionario es alguno de esos 3 elementos mencionados, y el valor es el conjunto de comportamientos que se ejecutarán cuando eso ocurra.

La tabla de símbolos almacena la información de las distintas variables, pero eso no bastoy a para realizar las verificaciones correspondientes de esta etapa. Adicionalmente, a cada árbol de instrucción y de expresión (explicados con detalle en el informe de la Etapa 2), se le ha colocado una función check, la cual permite chequear si los elementos de esa instrucción o expresión cumplen con las condiciones de un programa de BOT bien formado.

A continuación las distintas verificaciones que hace el analizador en esta etapa:

1. Verifica si hay errores de tipo en expresiones o contextos específicos. 1+true no es una expresión válida por ejemplo, y genera un error de tipo. 1 +

2 es una expresión válida, pero no es válida como guardia de una instrucción if o while, así que eso también generaría error de tipo. Al igual que el uso de store o collect sobre un tipo de elemento distinto al del robot que está siento declarado.

2. Verifica si las variables usadas están declaradas. Para hacer uso de, por ejemplo, la variable bot5, es necesario que bot5 haya sido declarada en algún punto anterior del programa. Nótese que subsecuentes declaraciones de la misma variable en alcances distintos esconden declaraciones anteriores, y subsecuentes declaraciones de la misma variable en el mismo alcance causan error por redeclaración.

3. Verifica el uso de la variable "Me", la cual solo puede ser usada durante la declaración de un robot, en sus comportamientos. El uso de "Me.$^{\text{en}}$ el lado .$^{\text{Ex}}$ecute"del programa, causará un error.

4. Verifica que dentro de los comportamientos de un robot no se haga uso de otros robots declarados o por declarar. Entre los comportamientos del robot, solo es posible el uso de la palabra "Me"para referirse al mismo robot que se está declarando.

5. Verifica también que solo exista la declaración de un comportamiento por robot. Es decir, para cada robot, a lo sumo puede existir un conjunto de instrucciones por comportamiento, subsecuentes declaraciones de comportamientos existiendo ya dicho comportamiento resultarán en un error. Adicionalmente, se verificará también que el comportamiento "default"sea el último en ser declarado.

Es importante mencionar que para la realización del análisis de contexto para esta etapa, se hace durante el parsing.

3. Revisión Teórico-Práctica

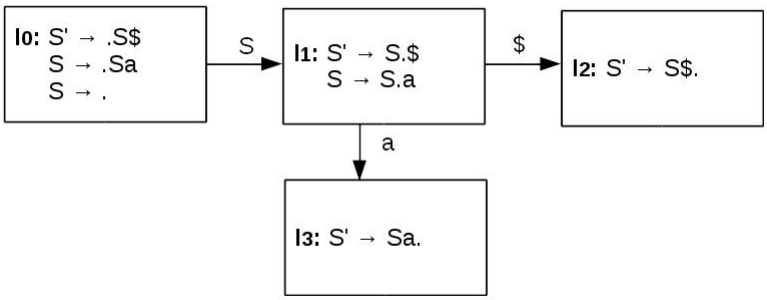
3.1. Pregunta 1

a) Para la gramática $G1_i$, tenemos:

1. $S' \rightarrow S\$$ (i)

2. $S \rightarrow Sa$ (ii)

3. $S \rightarrow \lambda$ (iii)



Existe un conflicto shift/reduce en I_0.

Items	**Acciones**		**Go to**	
	a	**$**	**S'**	**S**
I_0	reducir(iii)	reducir(iii)		1
I_1	avanzar(3)	avanzar(2)		
I_2		aceptar		
I_3	reducir(ii)	reducir(ii)		

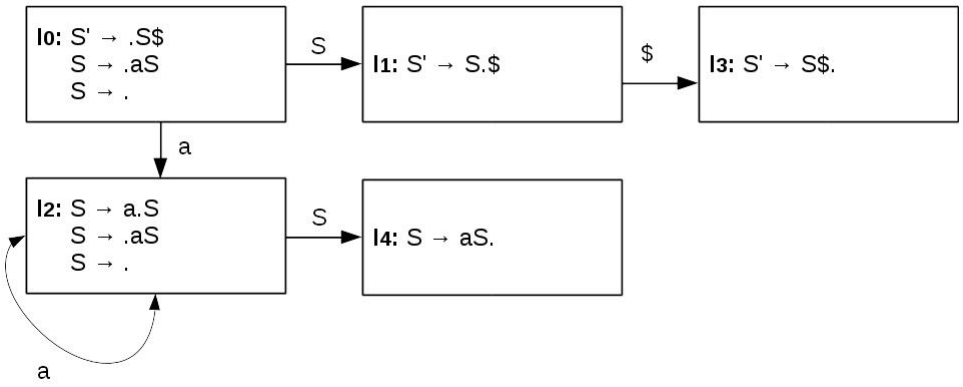
	FIRST	**FOLLOW**
S'	λ , a	$
S	λ , a	$, a

Para la gramática $G1_d$, tenemos:

1. $S' \rightarrow S\$$ (i)

2. $S \rightarrow aS$ (ii)

3. $S \rightarrow \lambda$ (iii)



Existe un conflicto shift/reduce en I_0 e I_2.

Items	**Acciones**		**Go to**	
	a	**$**	**S'**	**S**
I_0	avanzar(2)	reducir(iii)		1
I_1		avanzar(3)		
I_2	avanzar(2)	reducir(iii)		
I_3		aceptar		4
I_4	reducir(ii)	reducir(ii)		

	FIRST	**FOLLOW**
S'	a, λ	$
S	a, λ	$

b) Para comparar la eficiencia de ambos analizadores, decidimos realizar el reconocimiento de las frases a, aa, aaa, para las dos gramáticas, y poder observar un patrón y hacer el estudio en cuanto términos de espacio y de tiempo.

En términos de espacio, para la eficiencia de ambos analizadores, en el reconocimiento de a*, tenemos que la $G1_i$ parece ser más eficiente ya que hace un uso máximo de 3 items en la pila para cualquier cantidad de a's, en cambio, $G1_d$ para el ejemplo de aaa, hace un uso máximo de 4 items, por lo tanto $G1_i$ es más eficiente.

En términos de órdenes de complejidad, para el tiempo que se utiliza en reconocer frases de la forma a*, tenemos 2*n + 3 (siendo n la cantidad de a's de la frase) para $G1_i$, y para $G1_d$, es decir, para ambos casos el tiempo es O(n), es decir, es lineal.

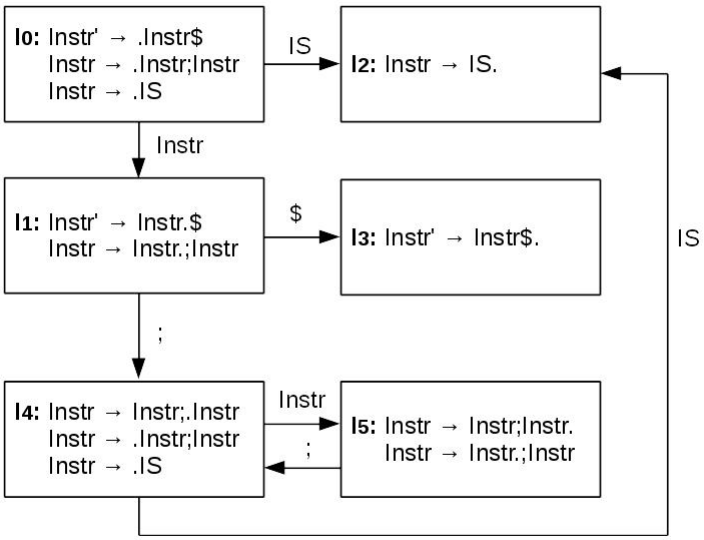
3.2. Pregunta 2

a)

1. $Instr' \rightarrow Instr\$$ (i)

2. $Instr \rightarrow Instr;Instr$ (ii)

3. $Instr \rightarrow IS$ (iii)



Existe un conflicto en I_5.

b)

Items	**Acciones**			**Go to**	
	;	**IS**	**$**	**Instr'**	**Instr**
I_0		avanzar(2)			1
I_1	avanzar(4)		avanzar(3)		
I_2	reducir(iii)	reducir(iii)	reducir(iii)		
I_3			aceptar		
I_4		avanzar(2)			5
I_5	avanzar(4) o reducir(ii)		reducir(ii)		

	FIRST	**FOLLOW**
Instr'	IS	$
Instr	IS	$, ;

El conflicto que existe es del tipo shift-reduce en I_5.

c) Para la primera alternativa, en favor del shift → teniendo avanzar(4):

Pila	**Entrada**	**Acción**
I_0	IS;IS;IS$	avanzar(2)
$I_2\ I_0$	;IS;IS$	reducir(iii)
$I_1\ I_0$	;IS;IS$	avanzar(4)
$I_4\ I_1\ I_0$	IS;IS$	avanzar(2)
$I_2\ I_4\ I_1\ I_0$	;IS$	reducir(iii)
$I_5\ I_4\ I_1\ I_0$	;IS$	avanzar(4)
$I_4\ I_5\ I_4\ I_1\ I_0$	IS$	avanzar(2)
$I_2\ I_4\ I_5\ I_4\ I_1\ I_0$	$	reducir(iii)
$I_5\ I_4\ I_5\ I_4\ I_1\ I_0$	$	reducir(ii)
$I_5\ I_4\ I_1\ I_0$	$	reducir(ii)
$I_1\ I_0$	$	avanzar(3)
$I_3\ I_1\ I_0$	$	aceptar

Para la segunda alternativa, en favor del reduce → teniendo reducir(ii):

Pila	**Entrada**	**Acción**
I_0	IS;IS;IS$	avanzar(2)
I_2 I_0	;IS;IS$	reducir(iii)
I_1 I_0	;IS;IS$	avanzar(4)
I_4 I_1 I_0	IS;IS$	avanzar(2)
I_2 I_4 I_1 I_0	;IS$	reducir(iii)
I_5 I_4 I_1 I_0	;IS$	reducir(ii)
I_1 I_0	;IS$	avanzar(4)
I_4 I_1 I_0	IS$	avanzar(2)
I_2 I_4 I_1 I_0	$	reducir(iii)
I_5 I_4 I_1 I_0	$	reducir(ii)
I_1 I_0	$	avanzar(3)
I_3 I_1 I_0	$	aceptar

Para la primera alternativa, se asocia hacia la izquierda, y en la segunda alternativa se asocia hacia la derecha, aunque se puede observar que hay ambiguedad en el lenguaje presentado, ya que se pueden crear dos árboles distintos para una sola gramática.

d) Para comparar la eficiencia de ambas alternativas, decidimos realizar más ejemplos, es decir, aparte de reconocer la frase de IS;IS;IS , reconocemos la frase IS;IS;IS;IS y la frase IS;IS;IS;IS;IS y vemos como se comportan.

Observamos un patrón donde, en términos de cantidad de pila, vemos que para la frase IS;IS;IS;IS empilamos un máximo de 8 items para shift, y para la frase IS;IS;IS;IS;IS empilamos un máximo de 10 items para shift. En cambio, para hacer reduce para la frase IS;IS;IS;IS y IS;IS;IS;IS;IS se empila un máximo de 4 items para ambos casos.

Entonces, viendo esto, para la frase IS;IS;IS que es la pedida, en términos de la cantidad de pila la segunda alternativa es mejor (abarca menos memoria), que serían con un máximo de 4 items. En cambio, en la primera alternativa se empilan más items que en la segunda, es decir, un máximo de 6 items. Viendo esto, tenemos 2*n + 1 , siendo n la cantidad de ;IS de la frase, para el máximo de memoria a usar en la pila.

En términos de órdenes de complejidad, para el tiempo que se utiliza para reconocer frases de la forma IS;$(IS)^n$, tenemos 4*(n + 1), es decir, es O(n) que es lineal, por lo tanto, tanto para el shift como para el reduce el tiempo es el mismo.

Referencias

[1] David M. Beazley - Documentación de Python-PLY http://www.dabeaz.com/ply/

[2] T. Sudkamp. Languages and Machines . Second Edition. Addison-Wesley, 1997.