



Trabajo Práctico

Generador de analizadores sintácticos

18 de junio, 2009

Teoría de lenguajes

Integrante	LU	Correo electrónico
Rodrigo Campos	561/06	rodrigo@sdfg.com.ar
Martín Fernandez	539/06/	bondi007@gmail.com
Matías Pérez	002/05	elmaildematiaz@gmail.com

Reservado para la cátedra

Instancia	Corrector	Nota
Primera entrega		
Segunda entrega		



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://exactas.uba.ar/>

Análisis del lenguaje de especificación de gramáticas

Tal como planteaba el enunciado de este trabajo práctico, el conjunto de producciones pertenecientes a la gramática que definirá el lenguaje para el cual finalmente se generará el analizador sintáctico pertinente, se encontraba expresado en un lenguaje que especifica producciones de gramática. Para dicho lenguaje se conocía, mediante el enunciado, su conjunto de producciones. Así, por ejemplo la cadena $A:aB \mid C$; escrita en el lenguaje mencionado, representaba la producción $A \rightarrow aB \mid C$, para cualquier gramática que la incluyera entre sus producciones. Dada esta situación, se tornaba fundamental comprender qué producción estaba representando efectivamente cada cadena escrita en el mencionado lenguaje de especificación. Además de esto, era indispensable poder verificar si dada una cadena escrita con los símbolos que define el lenguaje de especificación en cuestión, la misma pertenece o no al lenguaje. Ante estas necesidades, decidimos utilizar un analizador sintáctico llamado **pyparsing**, para el lenguaje de programación **python**. Este analizador nos ofrecía las funcionalidades requeridas y por ende nos permitía verificar si dada una cadena, la misma pertenecía a algún lenguaje para el cual se le especificaran al mismo las reglas de construcción de las producciones asociadas a su gramática. Sin embargo, en el momento de intentar utilizar pyparsing para analizar el lenguaje de especificaciones nos encontramos con que el conjunto de producciones dado en el enunciado que definía la gramática de dicho lenguaje, conformaba una gramática ambigua. Por ende, antes de proseguir con el desarrollo, nos encontramos ante la necesidad de desambiguar la gramática definida en el enunciado de este trabajo práctico. Con lo cual pasamos de tener inicialmente el siguiente conjunto de producciones:

- $G \rightarrow P \mid G \mid P$
- $P \rightarrow m \mid ' ' PD ' '$
- $PD \rightarrow n \mid t \mid \lambda \mid PD \mid PD \mid PD \mid ' * ' \mid PD \mid ' + ' \mid PD \mid ' ? ' \mid ' (PD) '$

(en donde t representa a un caracter en minúscula, que a su vez representa un símbolo terminal en la gramática que se intenta definir. El comportamiento de n es análogo, pero respecto a los no terminales de la gramática a definir, para lo cual n es un caracter en mayúscula).

a tener finalmente el conjunto de producciones desambiguado que se presenta a continuación

- $G \rightarrow \text{Prod} (\text{OtrasProd})^*$
- $\text{Prod} \rightarrow n \mid ' ' \text{ProdDer} ' '$

Una vez desambiguada la gramática, podíamos utilizar finalmente pyparsing para analizarla. Ahora, nos resultaba bastante sencillo determinar si una cadena representaba o no una producción válida construida en base a la gramática del lenguaje de especificación de gramáticas. Sin embargo, todavía nos quedaba la ardua tarea de generar el grafo con el cual trabajarían los algoritmos de cálculo de **anulables**, **primeros**, **siguientes**, y **símbolos directrices**, que son explicados posteriormente. En un principio creímos que podríamos realizar ambas tareas ("parsear" la gramática inicial y armar el grafo correspondiente a la gramática definida por la gramática inicial) en forma secuencial. Es decir, primero utilizar pyparsing para verificar que cada producción definida estuviera bien construida y perteneciera al lenguaje y luego en base a un algoritmo implementado por nosotros construir el grafo en cuestión. Lamentablemente, el intento resultó fallido, dado que la implementación de dicho algoritmo debía contemplar una gran cantidad de casos particulares, y resultaba además de "endeble", complicado de explicar y de entender y extremadamente complejo a la hora de realizar tareas de test y posterior "debug". Con lo cual, decidimos encarar la solución de este problema por otro camino. Tomamos la decisión entonces de revisar la documentación de pyparsing con el objetivo de verificar si el mismo no nos podía "dar una mano" con la construcción del grafo. Finalmente, descubrimos que pyparsing permitía realizar llamados a funciones auxiliares durante el proceso de análisis de una cadena. De esta manera, cada vez que encontrábamos un operador (como ser $' * '$, $' + '$, $' ? '$), un conector (como ser $' | '$, o implícitamente un conector de concatenación), un símbolo no terminal, o uno terminal podíamos llamar a la función auxiliar correspondiente a cada uno de ellos que se encargaba de, en cada caso, modificar o seguir construyendo (o ambas simultáneamente) el grafo representativo de la gramática en cuestión.

Breve reseña acerca de pyparsing

Pyparsing es un analizador sintáctico que, como tal, permite enunciar un conjunto de reglas sobre las cuales se construyen las cadenas válidas de un lenguaje. Estas reglas guardan gran similitud con las producciones que usualmente se utilizan para describir el comportamiento de una gramática. Mientras que en el lenguaje natural de gramáticas el símbolo \rightarrow se utiliza para denotar la producción derecha que genera un símbolo no terminal, en pyparsing este símbolo se representa con $=$, con lo cual una producción de la forma $A \rightarrow aB$, en pyparsing se reescribiría como $A = a + B$ (en donde el símbolo $' + '$ representa la concatenación en el lenguaje usual de producciones). Por otra parte, el operador $' | '$ se representa en pyparsing con el mismo símbolo. Además, el valor λ es representado con la sentencia **Empty()**. La funcionalidad del símbolo $' * '$, por otra parte, es representada por

la función **ZeroOrMore(cadena)**, mientras que las acciones del símbolo '+' son llevadas a cabo por la función **OneOrMore()**. Por ser python un lenguaje de programación, no podemos enunciar la producción $A = a + B$ sin haber definido antes que es 'B'. Esto puede ocasionar un problema si el símbolo inicial de la gramática tiene en su producción un no terminal todavía no definido. Para solucionar esto, alcanza con reemplazar a 'B' en la producción correspondiente, por alguna variable auxiliar, por ejemplo de nombre **otraB**, en donde otraB es una instancia de la clase **Forward**. Para la misma, en su inicialización se puede describir cual es el comportamiento que se espera que cumpla (en este caso, generar 'B').

Otras funciones que resultaron de utilidad a la hora de utilizar pyparsing son:

- **setParseAction(nombreDeFuncion)**, que fue utilizada en el momento de la creación del grafo correspondiente a la gramática para la cual se desea generar el analizador sintáctico. Así, por ejemplo, **OneOrMore(valor).setParseAction(esConcatenar)** invocará a la función **esConcatenar** cuando ocurrir que la cadena valor aparezca una o mas veces consecutivamente.
- **Literal(cadena)**, que permite definir distintos símbolos a utilizar durante el análisis.

Comprobar que la gramática sea ELL(1)

Para asegurarnos que la gramática sea ELL(1), hacemos dos chequeos en dos momentos distintos.

Antes de hacer cualquier chequeo, reducimos la gramática (es decir, hacemos que todos los no-terminales sean alcanzables y activos). Luego, por el teorema visto en clase que dice:

- Para toda gramática G reducida (osea, una en la cuales todos los no-terminales son alcanzables y activos), si G es independiente de contexto y LL(1), entonces G no es recursiva a izquierda.

Checkeamos si tiene recursión a izquierda, y si es así, concluimos que no es ELL(1).

Es decir, suponemos que la gramática es ELL(1). Sabemos entonces que no tiene recursión a izquierda. Si al chequear vemos que tiene recursión a izquierda, entonces llegamos a un absurdo que vino de suponer que era ELL(1). Es decir que podemos concluir que la gramática no es ELL(1).

Para poder aplicar el teorema anterior es necesario ver que la gramática es independiente de contexto. Para ver esto, podemos ver cómo se reescribiría cada símbolo que agregan las expresiones regulares en producciones de una gramática independiente de contexto y ver entonces que cada gramática cuyas producciones son expresiones regulares se puede transformar en una gramática independiente de contexto que genera el mismo lenguaje.

- Por cada expresión $A \rightarrow B \mid C$, se puede transformar en dos producciones: $A \rightarrow B$ y $A \rightarrow C$.
- Por cada expresión $A \rightarrow B^*$, se puede transformar en: $A \rightarrow BA$ y $A \rightarrow \lambda$
- Por cada expresión $A \rightarrow B^+$, se puede transformar en: $A \rightarrow BA'$ y $A' \rightarrow B$ y $A' \rightarrow \lambda$
- Por cada expresión $A \rightarrow B^?$, se puede transformar en: $A \rightarrow B$ y $A \rightarrow \lambda$

Es fácil ver que ambas expresiones tienen el mismo lenguaje.

Para transformar una expresión regular cualquiera, lo que se hace es por cada 'B*' que tenga, se puede introducir un nuevo no-terminal A, tal que $A \rightarrow BA$ y $A \rightarrow \lambda$, y reemplazar 'B*' en la expresión regular por A. Haciendo lo mismo para el resto de los símbolos siguiendo las transformaciones explicadas arriba, se puede ver que cualquier expresión regular se puede transformar en una gramática independiente de contexto equivalente.

Se puede notar también, de la forma de transformar las expresiones regulares, que si la expresión regular tenía recursión a izquierda, entonces luego de transformarlas también tendrán.

Es decir, si $A \rightarrow B^*$ tiene recursión a izquierda, entonces es $A \rightarrow A^*$, y las producciones generadas serán $A \rightarrow AA$ y $A \rightarrow \lambda$, es decir que hay recursión a izquierda. Es fácil ver que para el resto de los casos sucede lo mismo.

El segundo chequeo para ver que sea ELL(1), es el que vimos en clase, que lo hacemos luego de calcular anulables, primeros, siguientes y los símbolos directrices. Es decir, chequeamos que para cada '|', los símbolos directrices de sus hijos sean disjuntos y para cada '+', '*', '?', chequeamos que los primeros del nodo hijo sea disjunto con los siguientes del nodo y que ningún hijo sea anulable.

Reducir la gramática

Para reducir la gramática, primero buscamos cuáles son los nodos útiles del grafo.

Para esto primero buscamos los nodos activos, recorremos el grafo partiendo desde el símbolo distinguido, y por cada nodo, si es un terminal, λ , '*' o '?', lo marcamos como activo (pues siempre producen algo, ese terminal o λ). A cada no-terminal o '|', lo marcamos como activo si alguno de sus hijos es útil. Y al '.' y '+' los marcamos como activo sólo si todos sus hijos son útiles. Esto lo hacemos hasta que recorramos el grafo entero sin marcar ningún nodo nuevo como activo.

De esta forma tenemos cuáles son los nodos activos del grafo. Luego, lo que hacemos es sacar todos los links que tenga un nodo a un nodo inactivo, partiendo desde el símbolo distinguido. Es fácil ver que esto es correcto, pues:

- Si el nodo es no-terminal o '|', entonces esos casos, que nunca producirían nada, se pueden ignorar.
- Si el nodo es un '*' o '+' y sus hijos no son activos, entonces lo reemplazamos por λ y sacamos el link a su hijo, ya que es lo único que pueden producir
- Si, en cambio, es un '.' (representación utilizada para la operación de concatenación) o un '+', el nodo está activo si todos sus hijos tienen la misma condición, por lo que si tienen un hijo inactivo ellos también lo son, y por tanto, el padre sacará el link hacia ellos. Si no llegamos al no-terminal partiendo desde el símbolo distinguido, el no-terminal era inalcanzable, por lo que lo podemos ignorar.

Luego de este procedimiento convertimos a los nodos inactivos en inalcanzables, ya que los "desligamos" de la componente conexa que contiene al símbolo distinguido. Una vez que tenemos esto lo único que hace falta es quitar los nodos inalcanzables, para esto simplemente primero los reconocemos recorriendo la componente conexa y a medida que lo hacemos generamos el conjunto de los nodos de la misma, para luego reemplazar al conjunto de nodos del grafo por este nuevo conjunto.

Finalmente, después de aplicar estos dos procedimientos, el grafo se corresponde al de una gramática reducida (pues todos sus símbolos no-terminales son activos y alcanzables)

Checkear la recursión a izquierda

Para analizar la recursión a izquierda, usamos una idea muy similar a la que utilizamos para realizar el cálculo de "primeros". Recorremos el grafo de la misma manera, las diferencias son:

- Si el nodo actual es un terminal o λ , lo ignoramos.
- Si es '|', '*', '?', '+' o un no-terminal, hacemos lo mismo que hace primeros y además, para cada hijo si es un no-terminal que no pertenece a `rec_iz` del nodo actual, lo agregamos y marcamos que cambió algo en esta iteración.
- Si es '.', hacemos lo mismo que hace primeros para cada hijo, solo que además también si el hijo es un no-terminal lo agregamos a `rec_iz` del nodo actual.

Como consecuencia de esto, al finalizar tendremos un diccionario, el cual dado un nodo nos dice con qué no-terminales puede comenzar (por esto es que es muy similar a primeros). Si algún nodo puede comenzar con él mismo, entonces tiene recursión a izquierda. Si, por el contrario, ninguno de ellos puede comenzar con si mismo, entonces no se tiene recursión a izquierda presente en la gramática.

Simbolos directrices

Para calcular los simbolos directrices, primero calculamos para cada nodo, los anulables, los primeros y los siguientes.

Luego creamos un diccionario que tenga una entrada por cada nodo del grafo cuyo valor sea los primeros del mismo, y si dicho nodo es anulable, le agregamos los siguientes del nodo.

Para todos estos algoritmos utilizamos la misma forma de recorrer el grafo: recorreremos el grafo hasta que en una recorrida entera del grafo no hayamos agregado nada a lo que estamos calculando (anulables, primeros o siguientes). Si no cambio nada en una recorrida completa del grafo, es porque entonces ya hemos calculado lo que queríamos calcular.

Anulables

En cada paso (o nodo) entonces, lo que hacemos es fijarnos si el nodo actual está en el conjunto de nodos anulables y sino y cumple ciertos requisitos, lo agregamos y marcamos que cambió algo en esta iteración.

Los requisitos dependen del nodo actual, si es un terminal, nunca es anulable, por lo que no lo agregamos. Si es '*', '?' o ' λ ', siempre es anulable por lo que lo agregamos. Si es un no-terminal o un '|', es anulable si alguno de sus hijos es anulable. Y si es un '.' o '+', es anulable si todos sus hijos son anulables.

Al terminar de recorrer el grafo, tenemos el conjunto de nodos anulables.

Primeros

En cada paso, lo que hacemos es fijarnos cierta relacion entre los primeros del nodo actual y sus hijos, si no la cumplen los agregamos y marcamos que cambió algo en esta iteración.

Si el nodo actual es un no terminal y el caracter no pertenece a los primeros del nodo actual, lo agregamos. Si es un λ , lo ignoramos. Si es un '|', '*', '?', '+' o un no-terminal, entonces nos fijamos que contenga a los primeros de todos sus hijos. Si no los contiene, los agregamos y marcamos que algo cambio en esta iteracion. Y si es '.', nos fijamos si contiene a los primeros de su primer hijo (el de más a la izquierda), si no los contiene lo agregamos y marcamos que cambió algo. Si el primer hijo es anulable, nos fijamos que contenga a los primeros del segundo hijo, si no los contiene los agregamos y marcamos que cambió algo en esta iteración. Y así con los sucesivos hijos hasta que no tenga más o encontrar uno no anulable.

Cuando hayamos recorrido el grafo entero sin cambiar nada, entonces sabemos que todas las condiciones se cumplen, por lo que es fácil ver que se ha calculado los primeros de cada nodo correctamente.

Siguientes

En cada paso, lo que hacemos es fijarnos cierta relacion entre los siguientes del nodo actual y sus hijos, si no la cumplen los agregamos y marcamos que cambió algo en esta iteración.

Si el nodo actual es un terminian o λ , lo ignoramos. Si es un '|', '*', '?', '+' o un no-terminal, entonces nos fijamos que los siguientes de todos sus hijos incluyan a los siguientes del nodo actual. Si no es así, los incluimos y marcamos que algo cambió en esta iteración. Si es un '.' nos fijamos que los siguientes del '.' sean tambien siguientes de su último hijo (el de más a la derecha), si no es así los agregamos y marcamos que cambió. Si el último hijo es anulable, entonces nos fijamos que los siguientes de '.' también sean siguientes del nodo anterior al último y lo agregamos si no los incluye. Y lo mismo con el anterior y así sucesivamente hasta llegar a uno que no es anulable. También, para cada dos hijos 'x' e 'y', consecutivos ('x' más a la izquierda que 'y'), nos fijamos que los siguientes de 'x' contengan a los primeros de 'y', si no es así los agregamos. Y si 'y' es anulable, nos fijamos que los siguientes de 'x' contenga a los a los siguientes de 'y'.

Cuando hayamos recorrido el grafo entero sin cambiar nada, entonces sabemos que todas las condiciones se cumplen, por lo que es fácil ver que se ha calculado los siguientes de cada nodo correctamente.

Generación de código

Para generar el código que parsee las cadenas del lenguaje de la gramática que se obtiene como entrada cumpliendo con los requisitos pedidos se decidió elegir el lenguaje C++. Para hacerlo se procede de la siguiente forma.

Por un lado se tienen dos archivos *ParserEll1.cpp* y *Utilitario.cpp* estos contienen un main general y estandar que lo único que hace es leer la cadena de entrada, setear un par de variables y llamar a la funcion "parsear()" que será generada en el archivo *codigoParser.cpp* que es el que se generará. El archivo *Utilitario.cpp* es el que contiene la lógica del match, que se encarga de ver si la letra pasada para realizar el match es la misma del *TC* (Token Corriente), en caso de serlo aumenta el *TC* y en caso contrario tira el error correspondiente.

Por otro lado se genera el código de parseo específico para cada gramática. Esto se realiza escribiendolo en el standard output, por lo que es necesario ejecutarlo de la siguiente manera: *python programa >"ParserEll1.cpp"* para redirigir la salida al archivo *ParserEll1.cpp*.

El programa lo que hace es primero obtener los diccionarios *siguientes* y *primeros*, y una vez que se obtienen se imprimen una serie de lineas que corresponden a ciertos include genericos de las librerias que usará. Y luego se imprime para cada nodo No-Terminal una función, que tiene como nombre *Porc_ <Nombre Nodo>()* y para escribir el cuerpo de la misma se recorre el subgrafo que define el nodo llegando hasta las hojas o los nodos no terminales que se encuentren, para cada hijo se escribe:

- Si es un Terminal se hace un *match(<NombreNodo>);*
- Si es un No Terminal se hace un *Porc_ <Nombre Nodo>()*
- Si es un "|" se hace una serie de *if* para cada uno de sus hijos preguntando si el *TC* está en los simbolos directrices de ese nodo, y dentro de cada if se coloca el cuerpo de los hijos y se sigue. Y se termina con un *else* para el caso de error en el que se encuentra un caracter inesperado.
- Si es un "." se escribe directamente el codigo de todos sus hijos en orden.
- Si es un "*" se escribe un *while* preguntando si en *TC* se encuentra en los Primeros de ese nodo.
- Si es un "+" se actua de manera análoga al "*" pero con un *do while* en vez del *while*.

De esta manera se escribe todo el codigo correspondiente a esta gramática.

Para compilar el código simplemente se debe compilar *ParserEll1.cpp* con los archivos *Utilitario.cpp* y *codigoParser.cpp* en el mismo directorio.

Para ejecutarlo con la cadena *cadena* se debe ejecutar (en Linux): *nombreDelEjecutable <<< "cadena"*.