Trabajo Práctico

Generador de analizadores sintácticos

18 de junio, 2009 | Teoria de lenguajes

Integrante	LU	Correo electrónico
Rodrigo Campos	561/06	rodrigo@sdfg.com.ar
Martín Fernandez	/	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/

Comprobar que la gramatica sea ELL(1)

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

Antes de hacer cualquier checkeo, 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 gramatica G reducida (osea, una en las cuales todos lo 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 checkear 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 expresions 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 expresion A ->B | C, se puede transformar en dos producciones: A ->B y A ->C.
- Por cada expresion A ->B*, se puede transformar en: A ->BA y A ->λ
- Por cada expresion A ->B+, se puede transformar en: A ->BA' y A' ->B y A' ->λ
- Por cada expresion A ->B?, se puede transformar en: A ->B y A ->λ

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 ->BA y A ->λ, 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 ->B* tiene recursion a izquierda, entonces es A ->A*, y las producciones generadas seran A ->AA y A ->λ, es decir que hay recursión a izquierda. Es fácil ver que para el resto de los casos sucede lo mismo

El segundo checkeo para ver que sea ELL(1), es el que vimos en clase, que lo hacemos luego de calcular anulables, primeros, siguientes y los simbolos directrices. Es decir, checkeamos que para cada '|', los simbolos directrices de sus hijos sean disjuntos y para cada '+', '*', '?', checkeamos que los primeros del nodo hijo sea disjunto con los siguientes del nodo y que ningun 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 simbolo 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 y 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 es un '.' o '+' ólo son activos si todos sus hijos lo son, por lo que si tienen un hijo inactivo ellos también lo son, por lo que el padre sacará el link hacia ellos. Y si no llegamos al no-terminal partiendo desde el simbolo 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 simbolo 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.

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

Checkear la recursion a izquierda

Para ver la recursion a izquierda, usamos una idea muy parecida a la que usamos para calcular 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 macarmos 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.

Entonces, al finalizar tendremos un diccionario que 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 ninguno puede comenzar con si mismo, entonces ninguno tiene recursión a izquierda.

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 nodo, y si el nodo es anulable, le agregamos los siguientes del nodo.

Para todos estos algoritmos utilizamos la misma forma de recorrer el grafo: recorremos 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 termian 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 incluímos 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 especifico 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 menera 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"*.