CALCULO EFICIENTE DE CONJUNTOS DE LOOKAHEADS LALR(1) EN MICROCOMPUTADORES

Arturo Montes Sinning
Rodrigo López Beltrán
Universidad de los Andes
Bogotá - *Colombia*

1. Introducción

Durante el primer semestre de 1983 se comenzó un proyecto de software, en el departamento de sistemas de la Universidad de los Andes, cuyo objetivo era producir un generador de analizadores sintácticos LALR para ser utilizado como herramienta didáctica en el curso de construcción de compiladores de nuestro currículum. Dadas las restricciones de uso del equipo del Centro de Cálculo de la universidad (en donde ya se disponía de un generador de ese estilo), era necesario implementar el sistema sobre los microcomputadores que soportan buena parte de la atención a estudiantes. Una primera versión [MON85], que se ha venido utilizando desde comienzos de 1985, funciona sobre microcomputadores con sistema operacional MS-DOS.

Las ideas básicas para el desarrollo del proyecto fueron tomadas del trabajo de DeRemer y Pennello[DeR82], pero el resultado final contiene algunas mejoras con respecto al cálculo de los LookAheads (Algoritmo 3.6), aparte de ciertas ideas acerca de información útil para la resolución de los eventuales conflictos que se presentan en una gramática (Sección 4). El documento original de este trabajo [MON85], no resalta claramente las virtudes antes mencionadas pues se pierde en lo intrincado del formalismo utilizado, que es el propuesto en [DeR82]. Este artículo aclara las ideas de [MON85] gracias al excelente formalismo expuesto por Park, Chang y Choe [PAR85]. En las secciones 2 y 3 se exponen los algoritmos de cálculo del autómata LR(0) y los LookAheads, respectivamente, junto con la

introducción del formalismo "a la Park". En la sección 4 se esbozan las ideas con respecto a la resolución de conflictos y en la sección 5 se presentan algunas conclusiones.

2. Generación del Autómata LR(0)

Para la mayoría de las definiciones que aparecen a continuación utilizaremos la terminología habitual (ver [AHO77]). Algunas, que son la base de los resultados presentados en el artículo, han sido tomadas de [PAR85] y [DeR79].

Definición 2.1

Las nociones de símbolo y de cadena se suponen conocidas. Un vocabulario V es un conjunto de símbolos. V^* denota el conjunto de todos las cadenas de símbolos de V. V^+ denota $V^* - \{ \underline{lambda} \}$, donde lambda es la cadena nula. La longitud de una cadena alfa la denotaremos por |alfa|. El primer símbolo de una cadena alfa lo denotaremos por Prim(alfa); la cadena que sigue al Prim(alfa) será Rest(alfa) (el resto de alfa); el último símbolo lo designaremos por Ult(alfa).

Definición 2.2

Si R es una relación, R^* denota la clausura reflexiva y transitiva de R, y R^+ denota la clausura transitiva de R.

Definición 2.3

Una gramática G, es una cuádrupla G=<N,T,S,P> donde:

- N es un conjunto de símbolos llamados no-terminales.
- T es un conjunto de símbolos llamados símbolos terminales.
- T es disyunto de N, $S \in N$, y $V = N \cup T$.
- P es un subconjunto de $N \times V^*$ llamado Producciones de G. Si $(A,w) \in P$ escribiremos A -> w donde A se denomina la parte izquierda y w la parte derecha de una producción.

A continuación establecemos las siguientes convenciones que serán usuales en este artículo a menos que se especifique lo contrario:

S,A,B,C, ...	$\in$	N
X	$\in$	V
t,a,b,c, ...	$\in$	T
..... x,y,z	$\in$	T^*
alfa, beta, gamma ..	$\in$	V^*.

Definición 2.4

$=>_d$ es una relación de V^* en V^* tal que si alfa $\in V^*$ y z $\in T^*$, entonces alfaAz $=>_d$ alfawz si y sólo si A -> w $\in$ P. Al subíndice d lo eliminaremos de la notación, y cuando aparezca el símbolo => entenderemos que se trata de $=>_d$. $=>^*$ y $=>^+$ denotan la clausura reflexiva y transitiva de la relación => y ambas se leerán "deriva".

Definición 2.5

A es un no-terminal anulable si y sólo si, A =>* lambda.

Definición 2.6

alfa es una forma sentencial si y sólo si, alfa ∈ V* y S=>* alfa. Se llamará una sentencia si alfa ∈ T*.

Definición 2.7

L(G), el lenguaje generado por G, es el conjunto de sentencias. En otras palabras, L(G) = { x ∈ T* | S =>+ x }.

Definición 2.8

G es una gramática reducida, si y sólo si, para toda forma sentencial S =>+ alfaAbeta, existe y ∈ T* tal que A =>* y.

Definición 2.9

Para una gramática arbitraria G, podemos construir una gramática aumentada G' que contiene la producción S' -> S# en donde S' ∉ N y # es un nuevo símbolo terminal. Ahora L(G')= L(G)# y además se garantiza que el símbolo S' no aparece en la parte derecha de ninguna producción.

En lo sucesivo, supondremos que todas las gramáticas son reducidas y aumentadas.

Definición 2.10

Una tripleta (A,alfa,beta) es un ítem, si y sólo si, existe A -> alfa beta ∈ P, donde (A,alfa,beta) ∈ N x V* x V*. El ítem (A,alfa,beta) será denotado por A -> alfa.beta . Si alfa=lambda lo llamaremos un ítem inicial y si beta=lambda lo llamaremos un ítem final y si alfa y beta son lambda lo denominaremos un ítem nulo.

Definición 2.11

Una Tabla es un conjunto de ítems.

Definición 2.12

Si S es una Tabla,
CLAUSURA(S)= S U { A -> .w| B -> alfa.A beta ∈ CLAUSURA(S)}.

Definición 2.13

Si S es una Tabla y X es elemento de V,

GOTO(X,S) = CLAUSURA({A -> alfa X.beta| A -> alfa.X beta ∈ S}).

Definición 2.14

Si S es una Tabla, KERNEL(S) es el mínimo K, subconjunto de S, tal que CLAUSURA(K) = CLAUSURA(S).

Definición 2.15

Sea S una Tabla, SUCESORES(S) = { GOTO(X,CLAUSURA(S)) ¦ X ∈ V }.

Definición 2.16

TP(G), el conjunto de tablas de parsing LR(0) de una gramática G, es

$$TP(G) = \{ CLAUSURA(\{S \rightarrow .S'\#\}) \} \cup \{ CLAUSURA(K) \mid K \in SUCESORES(K') \text{ y } K' \in TP(G) \}.$$

Definición 2.17

Sea A -> alfa.Xbeta un ítem. La función GEN se define como,

GEN(A -> alfa.Xbeta) = ∅ si X ∈ T
ó
A -> alfa.Xbeta es un ítem final.

GEN(A -> alfa.Xbeta) = { X ->.w ¦ X -> w ∈ P } en cualquier otro caso. Por extensión, si S es una Tabla, entonces

GEN(S) = U {GEN*(A -> alfa.Xbeta)¦A -> alfa.Xbeta ∈ S}.

Lema 2.1

Sea A -> alfa.Xbeta un ítem,

CLAUSURA({A -> alfa.X beta}) = GEN*(A -> alfa.Xbeta).

Del lema y de la definición anteriores se observa que el cálculo de la CLAUSURA de una tabla, se puede realizar calculando GEN*. Por otro lado, GEN nos sugiere una relación (que llamaremos I) entre los no-terminales:

B I C si y solamente si B -> Cgamma ∈ P.

Al grafo dirigido asociado con la relación I lo llamaremos el I-grafo.

Lema 2.2

Sea A -> alfa.Bbeta un ítem, entonces

GEN*(A -> alfa.Bbeta) = { C ->.w¦ B I* C }.

Definición 2.18

El Autómata LR(0) para una gramática G (que en adelante

llamaremos ALR(0)) es una cuádrupla ALR(0) = (Q,i°,SUCC,TP(G)) donde

- TP(G) es el conjunto de tablas de parsing de G.
- Q es un conjunto de índices de una enumeración de TP(G).
- i° es el índice del primer elemento de la enumeración de TP(G) y corresponde a CLAUSURA({S' -> .S#}).
- SUCC es una función de Q x V en TP(G) tal que

  SUCC(p,X) = GOTO(X,Cp)

  donde Cp denota el p-ésimo elemento en la enumeración de TP(G). De ahora en adelante cuando referenciemos un elemento Cp de TP(G) nos estaremos refiriendo al p-ésimo elemento de la enumeración de TP(G) y con Kp al KERNEL(Cp). Cuando no se introduzca ambigüedad, utilizaremos un índice p como sinónimo de Cp.

El siguiente algoritmo nos calculará TP(G) para una gramática G cualquiera:

Algoritmo 2.1

```
Begin
  TP(G) := {CLAUSURA({S' -> .S#})}      ;
  For Each K ∈ TP(G) tal que sus SUCESORES no hayan sido
     calculados dO_Loop
     For Each K' ∈ SUCESORES(K) tal que K' ∉ TP(G) dO_Loop
         TP(G) := TP(G) U { K' } ;
     End_Loop ;
     Marcar a K indicando que sus SUCESORES ya han sido
     calculados ;
  End_Loop      ;
End  ;
```

Si guardamos en una cola los elementos de TP(G) cuyos sucesores no han sido calculados, obtenemos una primera mejora y el algoritmo quedaría de la siguiente forma:

Algoritmo 2.2

```
QK  :  Es una cola de los KERNELs de los elementos de TP(G)
       cuyos SUCESORES no han sido calculados.

Begin
    QK := <{S' -> .S#}>     ;
    TP(G) := {CLAUSURA({S' -> .S#})}  ;
    WHILE Not(Empty(QK)) dO_Loop
      K := DeQueue(QK)          ;
      For Each K' ∈ SUCESORES(K) tal que K' ∉ TP(G) dO_Loop
          TP(G) := TP(G) U { K' } ;
          EnQueue(K',QK)    ;
      End_Loop  ;
    End_Loop    ;
End     ;
```

Se observa, que la operación de averiguar si una Tabla K se encuentra en TP(G), es determinante en la complejidad del

algoritmo. Por lo tanto, respondiendo a ella de una manera eficiente podemos minimizar el costo del cálculo de la tabla de parsing.

Definición 2.19

Sea A -> alfa.Xbeta un ítem, se define la función ESTADOS como,

ESTADOS(A -> alfa.Xbeta)={ p ∈ Q ¦ A -> alfa.Xbeta ∈ K_p }.

donde K_p es el KERNEL del p-ésimo elemento en la enumeración de TP(G).

Lema 2.3

Sea q ∈ Q, entonces

Kq = { A -> alfa.Xbeta ¦ q ∈ ESTADOS(A -> alfa.Xbeta }.

Definición 2.20

Sea Cp una Tabla de TP(G) y q ∈ Q. Diremos que q es un estado candidato de Cp si y solamente si Kq es un superconjunto de S.

Lema 2.4

q - Q es un estado candidato de S, si y sólo si,

Para todo A -> alfa.Xbeta ∈ S, q ∈ ESTADOS(A ->alfa.Xbeta).

Por el lema anterior tenemos que el costo de responder a la pregunta P ∈ TP(G) es lineal con respecto a #({q¦ q es un estado candidato de P}), que es lo mismo que #({q¦ Kp⊂Kq}).

3. Cálculo de los LookAheads

Definición 3.1

Sea q ∈ Q y alfa ∈ V^*, se extiende SUCC a cadenas de V^*

SUCC(q,alfa) = q si alfa es lambda,

SUCC(q,alfa) = SUCC(SUCC(q, Prim(alfa)), Rest(alfa)).

Definición 3.2

Sea q ∈ Q, A ∈ N. La pareja (q,A) es una "transición no terminal" si y sólo si SUCC(q,A) está definido.

Definición 3.3.

Sea q ∈ Q y A -> alfa.beta un ítem,

LA(q,A -> alfa.beta) = { t ∈ T¦S' => S# =>* gammaAsigma# =>*
gamma alfa beta sigma# =>*
gamma alfa betatteta =>* x,
x ∈ L(G) y SUCC(i°,gamma,alfa) = Cq }.

Definición 3.4

Sea q ∈ Q y A -> w ∈ P,

LA(q,A -> w) = LA(q,A -> w.).

Definición 3.5

Sea q ∈ Q y t ∈ T,

Reduce(q,t) = { A -> w ∈ P ¦ t ∈ LA(q,A -> w) }.

Definición 3.6

Sea alfa ∈ V*,

FIRST(alfa)= {t ∈ T¦ alfa =>* tw para algún w ∈ T*}.

Si alfa es anulable entonces lambda también pertenece a FIRST(alfa).

Definición 3.7

Sean L, M conjuntos de cadenas,

L @ M = L si lambda no pertenece a L
ó
L @ M = L U M en caso contrario.

Definición 3.8

Sean B y C - N,

PATH(B,C) = U { FIRST(betan...beta2 beta1) ¦ B0=B, Bn=C,
B0 -> B1beta1 ∈ P, B1 -> B2beta2 ∈ P, ...,
B_{n-1} -> $B_n$betan ∈ P }.

Definición 3.9

Sea p ∈ Q y alfa ∈ V*,

PRED(p,alfa) = { q ∈ Q ¦ p = SUCC(q,alfa) }.

Lema 3.1 [PAR85]

Sean p, q ∈ Q tales que q ∈ PRED(p,alfa1), A -> alfa1.alfa2 ∈ Cp, t ∈ LA(p,A -> alfa1.alfa2), A ≠ S. Entonces debe haber por lo menos un ítem B -> beta1.A'beta2 ∈ Kq que satisface la siguiente condición:

```
t ∈ { a ¦ a ∈ PATH(A',A) @ FIRST(beta2t),
        q ∈ PRED(p,alfa1), A' I* A,
        B -> beta1.A'beta2 ∈ Kq}.
```

Teorema 3.1 [PAR85]

Sea p ∈ Q y A -> alfa1.alfa2 ∈ Cp, con A ≠ S, entonces:

```
LA(p,A -> alfa1.alfa2)={t¦t ∈ PATH(A',A)@FIRST(beta2)@
                          LA(q,B -> beta1.A'beta2),
   q ∈ PRED(p,alfa1), A' I* A, B -> beta1.A'beta2 ∈ Kq}.
```

Colocando la ecuación anterior en un lenguaje más ameno,

```
LA(p,A -> alfa1.alfa2)=
 U { U { PATH(A',A)@FIRST(beta2)@LA(q,B -> beta1.A'beta2) ¦
         A' I* A, B -> beta1.A'beta2 ∈ Kq
       } ¦
     q ∈ PRED(p,alfa1) }.
```

Esta ecuación muestra explícitamente la idea principal del algoritmo para el cálculo de los LookAheads.

Algoritmo 3.1

```
Function LA(p,A -> alfa1.alfa2) Return Set Of  T  Is

  Begin
      LA  := Ø       ;
      For Each q ∈ PRED(p,alfa1)  dO_Loop
          For Each  B -> beta1.A'beta2 ∈ Kq,A' I* A dO_Loop
                LA := LA U PATH(A',A)    ;
                dO_When (lambda ∈ PATH(A',A))
                    LA := LA U FIRST(beta2)    ;
                    dO_When(lambda ∈ FIRST(beta2))
                         LA := LA U LA(q,B -> beta1.A'beta2);
                    End_dO     ;
                End_dO     ;
          End_Loop ;
      End_Loop      ;
  End  ;
```

Se observa, que el algoritmo anterior requiere una modificación para evitar llamadas recursivas al mismo estado con el mismo ítem. Para ello, construimos una relación que nos evite casos como el anterior, de tal manera que dicha relación nos permita, por así decirlo, construir un orden de evaluación para el cálculo de los LAs.

Tomando las ideas propuestas por DeRemer y Pennello [DeR79], redefinimos el conjunto FOLLOW de la siguiente manera:

Definición 3.10

Sea q ∈ Q y A ∈ N,

FOLLOW(q,A)={a|a ∈ FIRST(beta2)@LA(q,B -> beta1.A'beta2),
B -> beta1.A'beta2 ∈ Kq, A I* A'}.

Desprendiéndose el siguiente lema:

Lema 3.2 [PAR85]

Sean p,q - Q y A -> alfa1.alfa2 ∈ Cp,

LA(p,A -> alfa1.alfa2) = { a | a ∈ FOLLOW(q,A),
q ∈ PRED(p,alfa1) }.

Reescribiendo los FOLLOWs con base en los FOLLOWs,

Lema 3.3 [PAR85]

Sea p,q ∈ Q,

FOLLOW(q,A) = { a|a - FIRST(beta2)@FOLLOW(r,B),
B -> beta1.Abeta2 ∈ Kq,
r - PRED(q,beta1),
C -> gamma1.Bgamma2 ∈ Cr }.

Si lambda ∈ FIRST(beta2), tenemos FOLLOW(q,A) incluye al FOLLOW(r,B).

Definición 3.11

(q,A) INCLUDES (r,B) si y solamente si se cumple la condición anterior.

El cálculo de los LAs según este método envuelve evaluar la clausura reflexiva de la relación INCLUDES. El cáculo de los LAs para un ítem final A ->w. de un estado p, es la union de los FOLLOW(q,A) donde q ∈ PRED(p,w). Obsérvese que

FOLLOW(p,A) = U FOLLOW(q,B) tales que (p,A) INCLUDES* (q,B).

Algoritmo 3.2 [DeR82]

Construcción del grafo debido a la relación INCLUDES.

```
For Each q ∈ Q  dO_Loop
        For Each B -> beta1.Abeta2 ∈ Cq dO_Loop
            FOLLOW(q,A) := FIRST(beta2)   ;
            INCLUDES(q,A) := Ø ;
            dO_When(lambda ∈ FIRST(beta2))
                For Each r ∈ PRED(q,beta1)  dO_Loop
                   For Each C -> gamma1.Bgamma2 ∈ Cr dO_Loop
                        INCLUDES(q,A) := INCLUDES(q,A) U { (r,B) }  ;
                   End_Loop    ;
                End_Loop   ;
             End_dO    ;
        End_Loop  ;
    End_Loop   ;
```

Algoritmo 3.3 [DeR82]

Cálculo de los LookAheads usando FOLLOWs,

```
For Each p ∈ Q  dO_Loop
    For Each A -> w. ∈ Kp  dO_Loop
       LA(q,A -> w.) := Ø ;
       For Each  q ∈  PRED(p,w)  dO_Loop
          LA(q,A -> w.) := LA(q,A -> w.) U FOLLOW(q,A) ;
       End_Loop ;
    End_Loop    ;
End_Loop    ;
```

El problema del método de DeRemer y Pennello es que la relación INCLUDES llega a ser demasiado grande, debido a la cantidad de ítems iniciales que pueden hacer aparecer arcos de la misma (en especial cuando hay producciones de la forma A -> B). Podemos obtener una reducción sustancial de dicha relación si sólo adicionamos arcos a través de los ítems presentes en el kernel exclusivamente. Y según el método desarrollado por J.C.H. Park, K. M. Choe y C. H. Chang [PAR85] tenemos,

Lema 3.4 [PAR85]

Sean p,q ∈ Q y A -> alfa1.alfa2 ∈ Cp,

$$LA(p,A \to alfa1.alfa2) = \{\, a \mid a \in PATH(A',A)@FOLLOW(q,A'),\ q \in PRED(p,alfa1),\ A' \ I^{*}\ A,\ B \to beta1.A'beta2 \in Kq \,\}.$$

Lema 3.5 [PAR85]

Sean p,q ∈ Q ,

$$FOLLOW(q,A') = \{\, a \mid a \in FIRST(beta2)@PATH(B',B)@FOLLOW(r,B'),\ B \to beta1.A'beta2 \in Kq,\ r = PRED(q,beta1),\ B' \ I^{*}\ B,\ C \to gamma1.B'gamma2 \in Kr \,\}.$$

Si lambda ∈ FIRST(beta2) y lambda ∈ PATH(B',B), tenemos FOLLOW(q,A') incluye al FOLLOW(r,B').

El algoritmo para generar la relación INCLUDES quedaría:

Algoritmo 3.4 [PAR85]

```
For Each q ∈ Q  dO_Loop
    For Each B -> beta1.Abeta2 ∈ Kq dO_Loop
        FOLLOW(q,A) := FIRST(beta2)    ;
        INCLUDES(q,A) := Ø  ;
        dO_When(lambda ∈ FIRST(beta2))
            For Each r ∈ PRED(q,beta1)  dO_Loop
               For  Each  C -> gamma1.Bgamma2 ∈ Kr,  B' I*  B
               dO_Loop
                  FOLLOW(q,A') := FOLLOW(q,A') U PATH(B',B);
                  dO_When(lambda ∈ PATH(B',B))
                      INCLUDES(q,A) := INCLUDES(q,A) U { (r,B) }
                  End_dO   ;
               End_Loop    ;
            End_Loop   ;
         End_dO    ;
    End_Loop  ;
End_Loop   ;
```

Y el cálculo de los LookAheads,

Algoritmo 3.5 [PAR85]

```
For Each p ∈ Q  dO_Loop
    For Each A -> w. ∈ Kp  dO_Loop
       LA(q,A -> w.) := Ø    ;
       For Each q ∈ PRED(p,w), B -> beta1A beta2 ∈ Kq dO_Loop
           LA(q,A -> w.) := LA(q,A -> w.) U PATH(A',A) ;
           dO_When(lambda ∈ PATH(B',B))
              LA(q,A -> w.) := LA(q,A -> w.) U FOLLOW(q,A) ;
           End_dO   ;
       End_Loop ;
    End_Loop   ;
End_Loop   ;
```

Todos los algoritmos anteriores calculan los FOLLOW de las transiciones no-terminales y construyen completamente la relación INCLUDES. Si se desea implementar dichos algoritmos en un microcomputador sería deseable construir un orden de evaluación sobre dicha relación para ir desechando FOLLOWs que no se necesitarán más en el cálculo de los LookAheads.

Extendiendo la relación INCLUDES entre no-terminales,

Definición 3.12

Si A, B ∈ N,

A INCLUDES B si y solamente si existe un ítem de la forma B -> alfa.Abeta tal que lambda ∈ FIRST(beta).

Esta relación no depende del autómata sino de la gramática. Si

esta idea va a ser usada en el algoritmo 3.5, |alfa| > 0.

Definición 3.13

Si A ∈ N,

$$ITEMS(A) = U \{ U \{ B \rightarrow \underline{alfa}.A\underline{beta} \mid \underline{lambda} \in FIRST(\underline{beta}) \} \mid A \text{ INCLUDES } B \}.$$

Habiendo construido la relación INCLUDES para no-terminales sólo resta construir un orden de evaluación utilizando dicha relación. Recordando cómo sería un orden de evaluación usando un grafo de dependencia, tendríamos las siguientes restricciones sobre dicho orden, donde Orden(A) significa el orden de evaluación de los LAs para los ítems finales que tienen a A en su parte izquierda.

- Orden(B) < Orden(A) si y solamente si A INCLUDES* B y no B INCLUDES* A.
- Orden(B) = Orden(A) si y solamente si A INCLUDES* B y B INCLUDES* A. A y B tienen el mismo orden de evaluación si y sólo si A y B se encuentran en un ciclo de la relación INCLUDES.

De aquí encontramos que la relación INCLUDES entre las transiciones no-terminales de dos no-terminales A y B, no tiene necesidad de construirse, siempre y cuando ni A ni B tengan el mismo orden de evaluación. Si por ejemplo, existen (p,A) y (q,B) tales que (p,A) INCLUDES (q,B) no tenemos necesidad de construirla pues sabemos que Orden(B) < Orden(A) y por lo tanto los FOLLOWs de las transiciones no-terminales de B se han calculado. Simplemente, a FOLLOW(p,A) se le adiciona el FOLLOW(q,B).

Definición 3.14

Sea i la i-ésima posición en el orden de evaluación de los LAs,

$$Gi = \{ A \mid Orden(A) = i \}.$$

El algoritmo para el cálculo de los LAs usando un orden de evaluación quedaría,

Algoritmo 3.6

```
    For Each i in 1..Max Orden de evaluación dO_Loop
       (*  Inicialización de los FOLLOWs y construcción de
           INCLUDES para los no terminales en este orden *)
       For Each A ∈ Gi
       dO_Loop
           For Each  (p,A) ¦ SUCC(p,A) está definida
           dO_Loop
               FOLLOW(p,A) := Ø          ;
               INCLUDES(p,A) := Ø         ;
           End_Loop                      ;
           For Each  B -> alfa.A beta ∈ ITEMS(A)
           dO_Loop
               For Each  (q,B) ¦ SUCC(q,B) está definida
               dO_Loop
                  p := SUCC(q,alfa)  ;
                  FOLLOW(p,A)  :=  FOLLOW(p,A) U FIRST(beta)  ;
                  dO_When  (  beta  =>*  lambda )
(*Línea 1*)          For Each  r ∈ PRED(p,alfa)
(*Línea 2*)          dO_Loop
(*Línea 3*)               For Each  C ->beta1.B'beta2 ∈ Kr
(*Línea 4*)                         donde B' I* B
(*Línea 5*)               dO_Loop
(*Línea 6*)                   FOLLOW(p,A) := FOLLOW(p,A) U PATH(B',B) ;
(*Línea 7*)                   dO_When ( lambda ∈ PATH(B',B) )
                                 dO_When ( B' ∈ Gi )
                                   INCLUDES(p,A):=INCLUDES(p,A) U
                                                  { (r,B') }          ;
                                 Else_dO
                                   FOLLOW(p,A) := FOLLOW(p,A)   U
                                                  FOLLOW(r,B')        ;
                                 End_dO    ;
                              End_dO  ;
                           End_Loop  ;
                        End_Loop  ;
                     End_dO  ;
                  End_Loop ;
           End_Loop  ;
       End_Loop ;
       (* Evaluación de la relación INCLUDES para los no-
          terminales presentes en este orden *)
       (*  Cálculo de los LAs para los ítems finales de los no-
           terminales envueltos en este orden *)
       For Each    A  ∈  Gi
       dO_Loop
          For Each A -> w. ∈ Cp  dO_Loop
              LA(q,A -> w.) := Ø     ;
             For Each q ∈ PRED(p,w), B -> beta1A beta2 - Kq
             dO_Loop
                 LA(q,A -> w.) := LA(q,A -> w.) U PATH(A',A) ;
                 dO_When ( lambda ∈ PATH(B',B) )
                    LA(q,A -> w.) := LA(q,A -> w.) U FOLLOW(q,A) ;
                 End_dO   ;
             End_Loop ;
```

```
            End_Loop    ;
        End_Loop    ;
    End_Loop    ;
```

Este algoritmo es válido para las dos mejoras (DeRemer y Pennello y Park), pues, para la mejora de DeRemer y Pennello es suficiente eliminar desde la (*Línea 1*) hasta la (Línea 7*) y en el caso de Park calcular el PATH de acuerdo con la definición. La mejora básica de este algoritmo sobre el de Park está en la generación del orden de evaluación. Recordando las estadísticas de DeRemer y Pennello, para gramáticas prácticas (PASCAL, MODULA, etc) el número de no-terminales envueltos en ciclos en la relación INCLUDES es muy pequeno (PASCAL 7 no-terminales), y en nuestro algoritmo sólo se crea la relación INCLUDES para los no-terminales presentes en el orden de evaluación. Normalmente, el número de transiciones no-terminales en un ciclo de la relación INCLUDES (para transiciones no terminales) tiene una cota superior en el número de arcos que aparecen en la relación INCLUDES (entre los no-terminales) presentes en el mismo orden de evaluación. Si sumamos todas las transiciones no-terminales de los no-terminales presentes en el mismo orden de evaluación y multiplicamos por la cota superior anterior obtendremos una cota superior (un poco exagerada) del número de arcos presentes en la relación INCLUDES para este algoritmo.

4. Resolviendo Conflictos

En esta sección esbozaremos algunas ideas que pueden emplearse en la construcción de algoritmos que ilustren el por qué se han producido ciertos conflictos en un autómata. La información puede utilizarse entonces para intentar resolver los conflictos, pero el éxito en esta labor dependerá de la astucia del usuario para desembarazarse de ellos.

Definición 4.1

Sea $q \in Q$ y $t \in T$,

q tiene un conflicto con t si y sólo si, SUCC(q,t) está definido y |Reduce(q,t)| > 0 y/o |Reduce(q,t)| > 1. Se llamará conflicto SHIFT-REDUCE si SUCC(q,t) está definido y Conflicto REDUCE-REDUCE si |Reduce(q,t)| > 1. Un conflicto a la vez puede ser SHIFT-REDUCE o REDUCE-REDUCE.

En el caso de un conflicto SHIFT-REDUCE debemos mostrar dos tipos de información, una debido a la componente del conflicto SUCC(q,t), que la denominaremos SHIFT-TRACE y otra debido a cada una de las producciones presentes en Reduce(q,t), que la denominaremos REDUCE TRACE.

Definición 4.2

Sea $q \in Q$, A -> alfa.beta $\in Cq$, decimos

CI(q,A -> alfa.beta) = { gamma ∈ V* | SUCC(i°, gamma alfa) = q }.

Sea q un estado que tenga un conflicto SHIFT-REDUCE con t. En qué consiste el SHIFT-TRACE ?. Si SUCC(q,t) está definido es porque existe por lo menos un ítem de la forma A -> alfa.tbeta ∈ Cq. Entonces, hay que justificar por qué A -> alfa.tbeta ∈ Cq, por lo tanto el SHIFT-TRACE es igual a CI(q,A -> alfa.tbeta) para cada ítem de la forma A -> alfa.tbeta ∈ Cq.

Sin embargo, CI es un conjunto demasiado grande (puede ser no finito), y a veces no es necesario presentar toda la información para entender el por qué del conflicto. Por ejemplo, presentando un elemento de CI que tenga la mínima longitud es suficiente (el contexto izquierdo más corto).

A continuación presentaremos un algoritmo que calcula el contexto izquierdo más corto, usando exclusivamente la función SUCC de un autómata ALR(0) cualquiera. Suponiendo que la numeración de los nodos del grafo asociado a dicho autómata se hubiera recorrido por extensión. La manera como se generó TP(G) en el algoritmo 3.2 nos permite obtener dicha numeración.

Definición 4.2

Sea alfa ∈ V* y p - 0,

MINPRED(p,alfa) = p si alfa es lambda
y
MINPRED(p,alfa) = MINPRED(MIN({ q|q ∈ PRED (p,Prim(alfa))}), Rest(alfa)).

Definición 4.3

Sea p,q ∈ Q y X ∈ V,

ROTULO(q) = X si y sólo si SUCC(p,X) = q.

Teniendo las definiciones anteriores,

Algoritmo 4.1

```
Function   CIC(p,A -> alfa.beta) return V*  is

Begin
    CIC  := lambda          ;
    q := MINPRED(q,alfa)    ;
    WHILE  q  ≠ i°  dO_Loop
        CIC := Append(CIC,ROTULO(q)) ;
        q  :=  MINPRED(q,ROTULO(q))  ;
    End_Loop            ;
    return  (CIC)       ;
End CIC ;
```

Definición 4.4

Sean B,C ∈ N, t ∈ T, t ∈ PATH(B,C) y, por la definición de PATH(B,C), existen B0=B .., Bn=C y B0 -> B1beta1 ∈ P, B1 -> B2beta2 - P, ..., Bn-1 -> Bnbetan ∈ P y t ∈ FIRST(betan... beta1). Por último, existiría un i ∈ [0..n] tal que t ∉ FIRST($beta_n$...$beta_{i+1}$) y t ∈ FIRST($beta_i$).

A,

- betan betan-1 ..... betai+1 =>* lambda, los llamaremos los lambda PATH(B,C) de t para todo i que satisfaga la anterior condición ;
- betai =>* t betai', los llamaremos las FIRST-PATH(B,C) derivaciones de t para lambda PATH(B,C) de t ;
- B0 =>* B1 betan .... beta1 la PATH(B,C) derivación,
- B0 -> B1 beta1 la producción contribuyente.

los llamaremos PATH-TRACEs de t en PATH(B,C).

Definición 4.5

Sea q ∈ Q, A ∈ N, t ∈ T, decimos que,

beta =>* tbeta'

Es una FIRST derivación de t en FOLLOW(q,A) si y sólo si existe B -> alfa.Abeta - Kq, tal que t - FIRST(beta).

El siguiente algoritmo nos devolverá todos los B -> alfa.Abeta ∈Kq que tengan una FIRST derivación de t en el FOLLOW(q,A).

Algoritmo 4.2

```
Function  FIRST_deriv(q,A,t) return N x V* x V* Is

    Begin
        FIRST_deriv :=  Ø        ;
        For Each  B -> alfa.Abeta ∈ Kq ! t ∈ FIRST(beta)
        dO_Loop
           FIRST_deriv := FIRST_deriv U { B -> alfa.Abeta } ;
        End_Loop  ;
        return (FIRST_deriv)        ;
    End   ;
```

Definición 4.6

Sea q ∈ Q, A ∈ N, t ∈ T, y B -> alfa.Abeta ∈ Kq, tal que t ∉ FIRST(beta), lambda ∈ FIRST(beta) y B' I* B tal que t ∈ PATH(B',B),

A los PATH-TRACEs de t en PATH(B',B) los denominaremos las PATHs derivaciones de t en FOLLOW(q,A) y a B -> alfa.Abeta el ítem contribuyente.

Algoritmo 4.3

```
Function   PATH_deriv(q,A,t) return ( NxV*xV* )xN Is

Begin
    PATH_deriv := Ø          ;
    For Each  B -> alfa.Abeta ∈ Kq ¦ beta =>*
    dO_Loop
       PATH_deriv := PATH_deriv U
                     { (B -> alfa.Abeta,{B'¦ t ∈ PATH(B',B)}) }
    End_Loop  ;
    return (PATH_deriv)      ;
End ;
```

La anterior función halla los ítems contribuyentes con sus respectivos no terminales que a través del PATH adicionan un símbolo terminal cualquiera al FOLLOW de la debida transición no terminal. Esta información es necesaria para producir todas las PATH-derivaciones de un terminal en el FOLLOW de una transición no terminal cualquiera.

Definición 4.7

Sea q ∈ Q, A ∈ N, t ∈ T, decimos que, FOLLOW(q,A) incluye directamente a t, si y sólo si, hay una FIRST derivación de t en FOLLOW(q,A) o hay una PATH derivación de t en FOLLOW(q,A).

Definición 4.8

(p,A -> w) LOOKBACK (q,B) si y solamente si existe B tal que lambda ∈ PATH(B,A), q ∈ PRED(p,w).

Si t ∈ LA(p,A -> w), entonces o existe B ∈ N tal que t ∈ PATH(B,A) o existen r y C tales que

(p,A -> w) LOOKBACK (q,B) INCLUDES* (r,C)

y FOLLOW(r,C) incluye directamente a t.

Ambos casos pueden suceder, pero nos interesa justificar la ocurrencia de al menos uno de ellos. Al primer caso, los vamos a llamar una inclusión directa de t en LA(p,A -> w); y al segundo, una inclusión indirecta de t en LA(p,A -> w). Las justificaciones de t - LA(p,A -> w) son todas las derivaciones de las inclusiones directas e indirectas y según las definiciones anteriores, se están abarcando todos los casos. Como siempre, para propósitos prácticos solo es deseable mostrar una de tales derivaciones: La primera que se encuentre según el método de búsqueda que sea empleado. No daremos algoritmos completos para la obtención directa de esta información, sino que nuestra intención es mostrar ideas generales para su desarrollo así como los algoritmos básicos para obtenerla.

5. Conclusiones

Las ideas desarrolladas en este trabajo, junto con las tomadas de los artículos descritos en la introducción y en la bibliografía, han sido hechas realidad en un sistema generador de analizadores sintácticos que ha sido usado como material de laboratorio en el curso de compiladores de la Universidad de los Andes (Bogotá, Colombia) con bastante éxito. La principal virtud del sistema es el tiempo de respuesta, debido en parte a la reducción de cálculos obtenida gracias al algoritmo 3.6.

El sistema, como se encuentra actualmente, recibe como entrada una gramática en forma de BNF o BNF-extendida y produce como salida un programa fuente manejador de la tabla de parsing, escrito en TURBOPASCAL, una descripción completa del autómata y las tablas, así como información para ayudar a resolver los conflictos.

Debido al excelente tiempo de respuesta y a la cantidad de información que proporciona, se ha convertido en una herramienta didáctica muy apreciada por los estudiantes, sobre todo por el hecho de ser utilizable sobre microcomputadores lo que le da una flexibilidad de acceso difícil de obtener sobre un computador grande (máquinas en las que tradicionalmente se ha implementado este tipo de software). Por otro lado, además de haber sido usado como material de laboratorio, el compiler-compiler ha sido probado para diferentes gramáticas "reales", entre las cuales tenemos: MODULA, TURBOPASCAL, un subconjunto de ADA (no hay problemas en la generación de ADA), una serie de gramáticas para generación de reportes y la gramática de PASCAL ANSI, con resultados satisfactorios, demostrando que el sistema es útil en un ambiente de producción real y es algo más que una facilidad didáctica.

En el estado actual, los fuentes están siendo pasados a C para obtener un mejor desempeño, aunque el tiempo de respuesta es excelente (sobre todo teniendo en cuenta que se utiliza en microcomputadores standard de 16 bits). Adicionalmente, se ha pensado agregarle otros módulos que comprenden: generador autómatico de tablas de símbolos, un generador automático de analizadores léxicos, un compactador de tablas esparcidas, un sencillo evaluador de atributos, un algoritmo general para generación de código para expresiones y un lenguaje para la especificación de lenguajes de máquina.

Referencias

[AHO77] Aho A, Ullman J.
"Principles of Compiler Design"
Addison-Wesley, Reading Mass., 1977.

[DeR82] DeRemer F, Pennello T.J.
"Efficient Computation of LALR(1) LookAhead Sets"
ACM TOPLS Vol. 4 No. 4, octubre 1982

[MON85] Montes A.
"Diseño e Implementación de un Generador de Compiladores"
Proyecto de Grado, Dpto. de Sistemas, UNIANDES, Bogotá, 1985.

[PAR85] Park J, Choe K, Chang C.
"A New Analysis of LALR Formalisms"
ACM TOPLS Vol. 7 No. 1, enero 1985.