Un Algoritmo Práctico Mejorado para Parsing de Lenguaje Natural basado en Gramáticas ATN No-deterministas

John A. Atkinson Abutridy
Departamento de Ingeniería Informática y Ciencias de la Computación
Universidad de Concepción
Concepción, Chile
atkinson@inf.udec.cl

Anita A. Ferreira Cabrera
Facultad de Educación y Artes
Universidad de Concepción
Concepción, Chile
aferreir@buho.dpi.udec.cl

Resumen

El presente trabajo considera un algoritmo de reconocimiento para implementar interfaces en lenguaje natural basadas en ATN [4]. El algoritmo presenta algunos mejoramientos con respecto de aquellos de funciones similares de Parsing. Entre ellos se incluyen el manejo de ambigüedad, facilidades de programación y post ejecución de acciones semánticas.

El algoritmo se implementó para construir una herramienta de software que permite construir automáticamente interfaces para lenguaje natural, de mediano tamaño. En suma, se describen los algoritmos diseñados, el sistema que utiliza dichos algoritmos y sus resultados.

1 Introducción

En el ámbito de las interfaces en lenguaje natural, el aspecto del parsing toma un rol protagónico junto con el proceso de análisis semántico, puesto que consideran una serie de filtros y se constituyen en una de las etapas más importantes involucradas en el reconocimiento del lenguaje. Existe una gran variedad de técnicas de reconocimiento. No obstante, la mayoría presenta problemas, entre ellos podemos citar los siguientes:

- Dependen de un lenguaje en particular y son específicos del dominio [9].
- Algunos mecanismos son demasiado complejos y poco prácticos al ser implementados.
- Asumen que la especificación gramatical de entrada es determinista.
- No existe un enfoque incremental en futuras implementaciones o modificaciones de las mismas.

- Los esquemas de almacenamiento del conocimiento léxico y sintáctico no permiten flexibilidad ni independencia entre las partes, de modo que futuras modificaciones en el corpus alteran enormemente el sistema computacional y lo hacen cada vez más complejo y difícil de mantener.

- El sistema de reconocimiento debe ser modificado cada vez que crece el corpus lingüístico.

El algoritmo de parsing que aquí se describe está basado en el modelo gramatical extendido especificado en forma de ATN [4, 2]. Con el objeto de lograr aplicaciones más prácticas y mejorar la potencia de los ATN se han incorporado algunos cambios que constribuyen a un mejoramiento tanto a nivel de la programación, como en lo que compete al manejo gramatical desde una perspectiva lingüística.

Las siguientes secciones describen el algoritmo investigado, las características de su implementación, sus aplicaciones y ventajas respecto de otros enfoques.

2 El Algoritmo

El ATN [11, 4] es por naturaleza un sistema recursivo en el cual se realizan computaciones y sub computaciones de los constituyentes de las sentencias, dependiendo de las categorías que estén involucradas en la gramática de entrada. Para efectos de nuestro enfoque, cada uno de los ATN de una gramática dada posee su propia tabla de transición similar a la de un autómata finito.

Por ejemplo, en la figura 2, se muestra la correspondiente tabla para el ATN *SV* dado en la figura 1.

Dichas tablas son muy fáciles de visualizar, mantener y accesar. Su generación es más simple que tablas del tipo goto-action utilizadas en parsers LR [1]. Una de las diferencias fundamentales es que los símbolos de las transiciones son más poderosos y permiten realizar subcomputaciones de una tabla a otra, manteniendo el estado global del sistema consistente y actualizado constantemente. Dicho tipo de arcos se convierte en condiciones que se deben cumplir para pasar de un estado a otro.

Si además a lo anterior se agrega el hecho de que en el proceso de análisis, el parsing requiere actuar en paralelo con el análisis semántico y otras fases, se genera el problema de manejar el no determinismo de una manera eficiente y transparente al usuario. Esto origina, en términos simples, que por ejemplo, existan varios caminos de análisis para una misma entrada, problema conocido en Parsing como *ambigüedad* [1]. Esto causa que al verificarse las condiciones en las transiciones de cada ATN y ejecutarse las acciones correspondientes, se pueda llegar en un momento en que las condiciones sean falsas y por lo tanto deberían deshacerse todas para probar otro camino. Ello resulta imposible o muy complejo en la mayoría de los algoritmos.

Otro problema de implementación relevante que existe, está referido a acciones del tipo `hold`, presentes en un ATN. Estas dependen de la satisfacción de una condición `vir`. Ahora bien, pero qué ocurre si por esta razón se realiza la acción semántica del bloque en el cual está `hold`?, en este caso se hace imposible deshacer todo, si es que posteriormente alguna condición no se cumple.

Esta situación es cubierta por el algoritmo propuesto de la siguiente forma: Cada uno de los bloques de acciones semánticas de un ATN se identifica por un índice correlativo simple (`Actid`). Una vez que se satisface una condición, se almacena su identificador en una cola interna. Si se llega correctamente al final del análisis, se ejecutan secuencialmente todas las

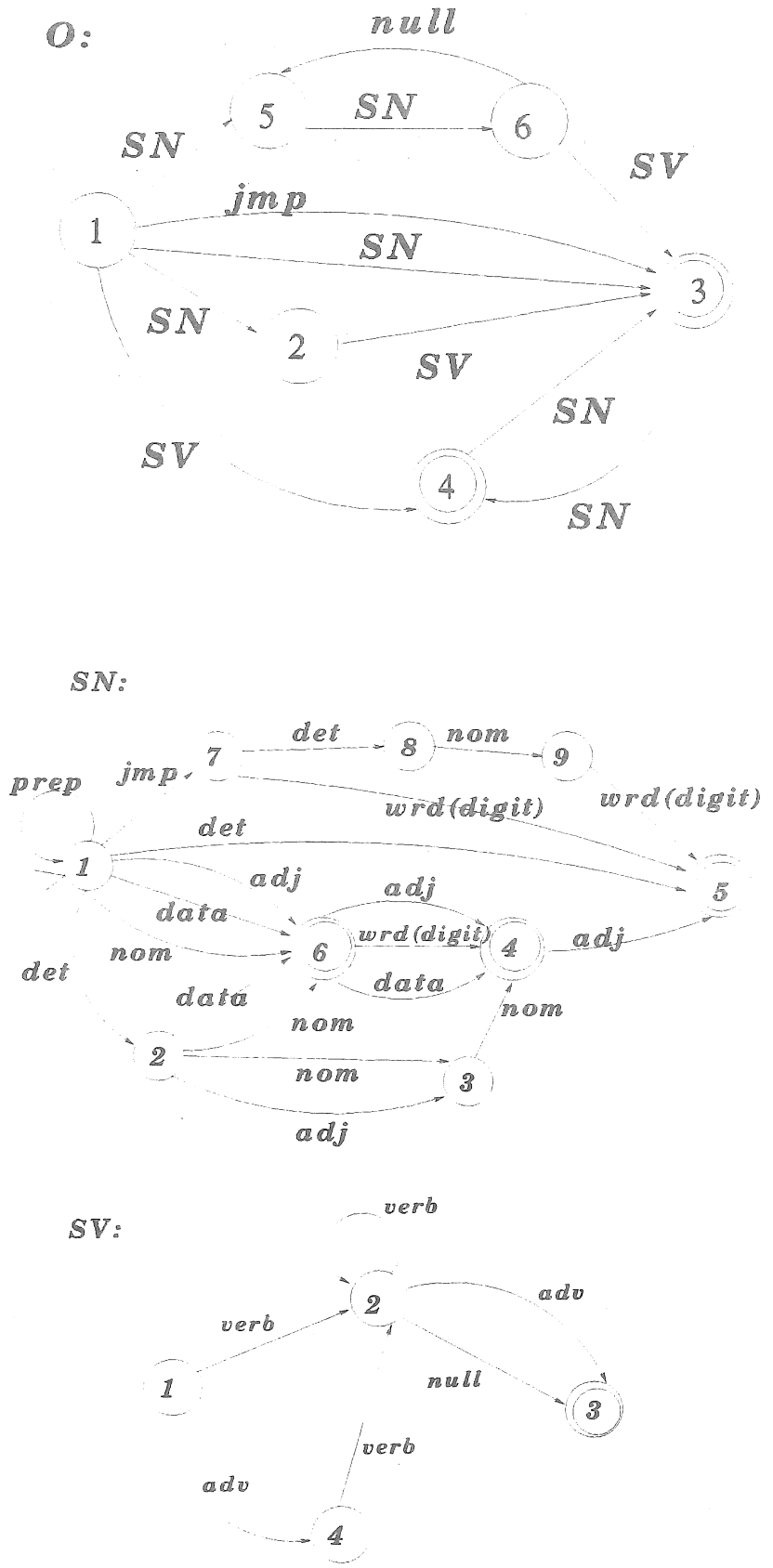

Figura 1: Un ATN no-determinista

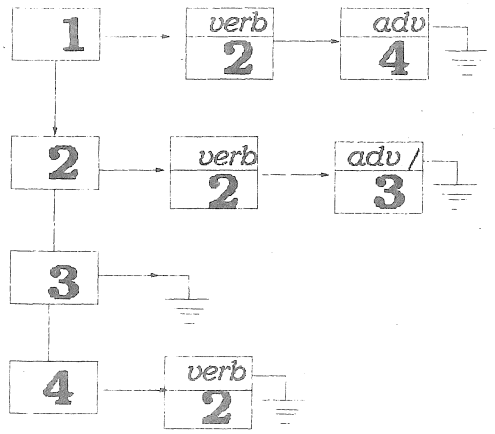

Figura 2: Tabla de transición para un ATN simple

acciones almacenadas. Por el contrario, si en un cierto ATN una condición no se cumple, sencillamente se "deshace" la acción, eliminando su identificador del fin de la cola. Por tanto, el funcionamiento de esta cola extendida se basa en tres tareas posibles:

- Extraer un ActId del frente de la cola (si la acción se va a ejecutar).
- Remover un ActId del fondo de la cola (si la acción se va a deshacer).
- Agregar un ActId al fondo de la cola (si se atraviesa un arco exitosamente)

Cabe notar que las acciones se ejecutan automáticamente una vez que finaliza la computación del ATN inicial, de la siguiente forma:

```
MIENTRAS Cola.no.vacia HAGA
        DEQUEUE(Cola, ActId)
        EJECUTAR.ACCION(ActId)
FIN-MIENTRAS
```

Adicionalmente, el algoritmo de parsing mantiene en stack locales los estados pendientes por los que está atravesando con el fin de realizar el proceso de backtracking cuando ciertos arcos fallan o existe ambigüedad, por lo cual continuamente se están procesando stacks y colas en forma dinámica cada vez que se procesa una transición [2].

Para el problema relacionado con acciones `hold` [4, 2], previamente un preprocesador extrae del bloque correspondiente la llamada hold, de modo tal que se pueda ejecutar independientemente del resto, de este modo en un caso determinado se puede deshacer las acciones.

El algoritmo implementado que interactúa con otros módulos de análisis y ejecución, es básicamente el siguiente:

```
ALGORITMO: DRIVER ATN

ENTRADA: Estado.inicial, ATN.id
SALIDA: Resultado.evaluacion.ATN
Num.de.acciones, Num.de.arcos

      BEGIN

            Q = estado.inicial
            CREATE(Stack)
            Marcar.Estado.Visitado(Q)
            Num.de.acciones=0
            Num.de.arcos=0
            REPEAT FOREVER
                 Buscar.Condicion.Verdadera(Q,ATN.id)
                 If encontro.condicion Then
                     Ejecutar.acciones.HOLD.de.arco.actual(ActId)
                     ENQUEUE(Q,ActId)
                     Num.de.acciones=Num.de.acciones+
                                 acciones.retornadas.por.arco
                     Num.de.arcos=Num.de.arcos+1
                     If (existen.varias.transiciones) Then
                         PUSH(Stack, Q, Num.de.arcos)
                     End-If
                     If (arco.consume.simbolo()) Then
                         Lea.Siguiente.entrada()
                     End-If
                     Marcar.Estado.Visitado(Q)
                 Else If estado.final(Q) Then
                           return(TRUE)
                     Else  If EMPTY(Stack) Then
                                 DESHACER.ARCOS(Num.de.arcos)
                                 return(FAIL)
                       Else
                                 Marcar.Estado.Visitado(Q)
                                 POP(Stack,Q,num.arcos.previos)
                                 DESHACER.ARCOS.ATRAVESADOS()
                            End-If
              End-Else
          End-Else
              End-If
       END-REPEAT
END-ALGORITMO
```

Debido a que el parser analiza un ATN y éste puede invocar a otros o bien a sí mismo, el resultado de una computación parcial es el estado, sea verdadero o falso, asumiendo que la llamada a un ATN puede ser una condición normal de algún arco. Además, con el fin de ejecutar posteriormente las acciones semánticas que correspondan, las otras salidas corresponden al número de arcos y acciones atravesadas. Es importante destacar que si una condición sobre un arco es una llamada recursiva a un ATN, las acciones serán las del propio arco más las acumuladas de la computación parcial de dicho sub ATN.

El procesamiento de una tabla de transición es muy similar al de un autómata finito. Existe como punto de partida un estado inicial (Q), dentro de un ATN inicial (ATN.id), un símbolo que viene del string de entrada y una condición que se debe satisfacer. Cada vez que se atraviesa un arco, se almacena en una estructura LIFO el estado actual de modo de poder manejar posibles estados de falla futuros y realizar el backtracking correspondiente.

Si encuentra una condición que se satisface desde el estado actual, se ejecutan las acciones hold del bloque de acción semántica respectiva (si es que existe). Se almacena en la cola el índice de dicha acción, luego se acumula el número de acciones que se realizó y el número de arcos atravesados para finalmente visitar el estado actual.

Si el algoritmo encuentra que hay más de una transición posible, toma una de ellas y apila el estado actual y el número de arcos acumulados para probar ese camino.

El hecho de no hallar una condición que satisfaga a partir del estado actual se puede deber a tres razones:

- Se llegó a un estado final: la computación termina y se retorna exitosamente al nivel de computación superior.
- El stack está vacío: significa que hubo un error, por lo cual se deshacen las acciones realizadas hasta el momento y la computación falla.
- El stack no está vacío: significa que hay estados pendientes por visitar (comienza el proceso de retroceso) por lo cual se debe probar otra alternativa factible y se desapila el último estado visitado, deshaciendo las acciones transcurridas desde ese estado hasta el actual.

3 Resultados

El algoritmo aquí descrito fue implementado y utilizado para construir una herramienta de software denominada GILENA, la cual permite generar interfaces en lenguaje natural a partir de especificaciones de entrada en la forma de ATN y extensiones que permiten facilidades de programación. La arquitectura básica del sistema se muestra en la figura 3.

La entrada al sistema corresponde a la especificación de la gramática, las características de su léxico y las acciones semánticas correspondientes. Por ejemplo, parte de la entrada al sistema, dado la gramática de la figura 1, es la siguiente:

```
atn

   0:

      s1 -> SN to s5 |
            jmp to s3 |
            SN  to s5 |
            SN  to s2 |
            SV  to s4;
      s2 -> SV to s3;
      s3 -> SN to s4 | ;
      s4 -> SN to s3;
      s5 -> SN to s6;
      s6 -> SV to s3 | ;

&
   SN:

      s1 -> cat(prep) to s1 |
            cat(adj) to s6 |
            cat(data) to s6 |
            cat(nom) to s6 |
            cat(det) to s2;

      s6 -> cat(adj) { <alguna accion> } to s4 |
            wrd(digit) to s4 |
            cat(data) { <alguna accion> }  to s4;
      ...
&
   SV:

      s1 -> cat(verb) to s2 |
            cat(adv) to s4;
      ...
 enda
```

Una de las salidas del sistema corresponde al parser cuyo algoritmo básico ya se ha mostrado. Las tabla de transición y las tablas de símbolos del intérprete utilizadas por el algoritmo son generadas automáticamente por GILENA [2].

Debido a que la salida del sistema corresponde a programas escritos en C, las acciones se indexan y se genera código para la ejecución de ellas, de tal manera de ser ejecutadas en el momento adecuado dando su identificador. El cuerpo general del módulo que se genera para ello es:

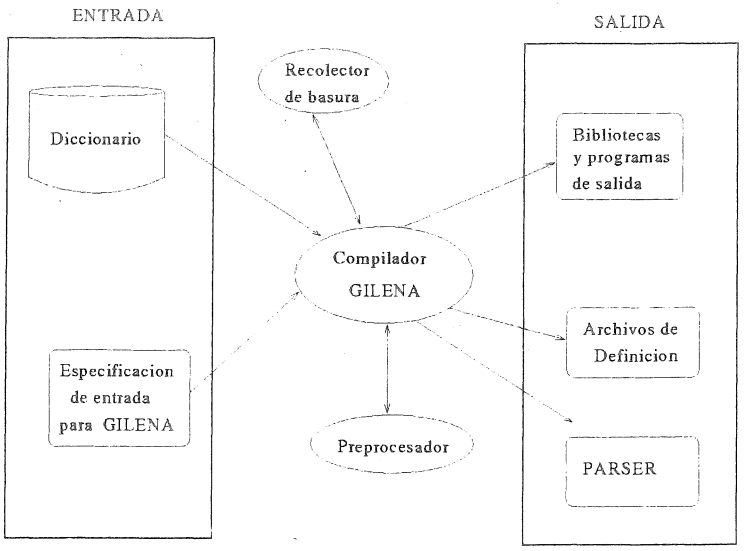

Figura 3: Arquitectura del sistema

```
ejecutar.accion(int ActId, int es.accion.hold)
{
    switch (ActId){
       case 0: if (es.accion.hold)
                    <acciones hold del bloque 0>
               else <acciones normales del bloque 0>
               break;
       case 1: if (es.accion.hold)
                    <acciones hold del bloque 1>
               else <acciones normales del bloque 1>
               break;
       ...
    }
 }
```

La normalización de símbolos, separación de acciones `hold` , y otras tareas, son llevadas a cabo por un módulo de pre-procesamiento integrado al sistema que actúa en conjunto con las tablas de símbolos.

El procesamiento de cada uno de los ATN de la gramática también requiere un manejo especial. Esto depende del ATN fuente y del ATN destino (o al que se llama en un momento dado). Desde el punto de vista del algoritmo, la lógica es la misma, sin embargo, debe precisarse cual es el ATN para determinar cual es la tabla de transición y símbolos que se utilizarán.

Para lograr lo anterior, la herramienta genera automáticamente las correspondientes llamadas al parser (Driver de ATN) con el identificador del ATN que corresponda, tal como se muestra a continuación:

```
probar.ATN(int ATN.fuente,int ATN.destino)
{
    switch (ATN.fuente){

       case 0: if (ATN.destino==1)
                    DriverATN(1, <estado inicial>)
               else if (ATN.destino==2)
                   DriverATN(2, <estado inicial>)
        ...
       case 1: if (ATN.destino==2)
                     DriverATN(2, <estado inicial>)
               else if (ATN.destino==3)
                    DriverATN(3, <estado inicial>)
               ...
    }
}
```

4 Ventajas

El algoritmo y su implementación, generada automáticamente por la herramienta GILENA, anteriormente descrita posee las siguientes ventajas:

- Debido al manejo del no determinismo del algoritmo, se facilita el hecho de expresar la gramática de entrada en forma más natural y sencilla para el usuario implementador de la interfaz.
- En la implementación actual, el manejo de las categorías gramaticales dentro de los *ATN* es independiente del léxico o contenido del diccionario de símbolos. Es decir, si una vez que una interfaz está operando, se desean hacer cambios, sólo se deben realizar los cambios a los diccionarios y no en los *ATN* de entrada, en consecuencia las reglas gramaticales no crecen proporcionalmente.
- Además de las características propias de los *ATN* [4, 2], se ha implementado una biblioteca de funciones que permiten al usuario recuperar y analizar información del léxico o diccionario, tener control sobre el contexto en las oraciones y manejar irregularidades lingüísticas. Todo lo cual es interpretado automáticamente por el algoritmo.
- Los *ATN* que el algoritmo de parsing utiliza son capaces de detectar símbolos con tratamiento especial, analizar automáticamente conjunciones o algúna otra característica, los cuales deben ser especificados en la entrada.

5 Conclusiones

En este trabajo presentamos un algoritmo y consideraciones para su posterior implementación. Como resultado, se han generado varios sistemas que utilizan un esquema de parsing basado en dicho algoritmo.

Nosotros probamos que para interfaces de mediana complejidad, el parser maneja bien el proceso de análisis y la determinación de los diferentes caminos posibles ante condiciones de ambigüedad. A diferencia de otros enfoques y algoritmos de parsing basados en ATN [8], el nuestro es independiente de las reglas gramaticales y del lenguaje (o sub lenguaje) elegido.

Aunque los modelos ATN presentan algunas limitantes intrínsecas para el análisis del lenguaje, su potencialidad fue mejorada permitiendo manejar reglas ambigüas de manera natural, post ejecución de acciones semánticas sin perder el efecto paralelo del análisis Sintáctico/Semántico.

Como resultado, el algoritmo sirvió como pieza fundamental para el desarrollo de la herramienta GILENA, la que permite generar en forma automática interfaces en lenguaje natural.

Comparado a otros algoritmos, el nuestro es fácil de implementar, modificar y puede crecer gradualmente para manejar situaciones de análisis más complejas, de una manera sencilla.

Debido a la forma en que se implementó el parser, la herramienta GILENA es capaz de realizar el seguimiento (*debugging*) que realiza el algoritmo en cualquier punto, a demanda del usuario.

Referencias

[1] Aho, Alfred & Ullman, Jefrey. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Atkinson, John. *GILENA: Generador de Interfaces en Lenguaje Natural.* B. Eng. Thesis, Departamento de Informática, Universidad Técnica Federico Santa María, Valparaiso, Chile, Diciembre 1991.

[3] Atkinson, John. *A Flexible New Language for Specifying and Generating Natural-Language Systems.* Proceedings of the Third Natural Language Processing Pacific Rim Simposium, pp. 612-615, Seoul, Korea (December 1995).

[4] Bolc Leonard. *The Design of Interpreters, Compilers and Editors for Augmented Transition Networks.* Springer Verlag, 1986.

[5] Ferreira Anita. *Construcción de un Analizador y Sintetizador Computacional de oraciones subordinadas castellanas.* MA Thesis, Facultad de Educación, Humanidades y Artes, Universidad de Concepción, 1990.

[6] Ferreira Anita. *Diseño e Implementación de un Analizador Funcional en Procesamiento de Lenguaje Natural.* Revista de Lingüística Aplicada (RLA), No. 30, 1992.

[7] Ferreira Anita. *El Lexicón en Procesamiento de Lenguaje Natural.* Revista de Lingüística Aplicada (RLA), No. 34, 1994.

[8] Seneff, Stephanie. *TINA: A Natural-Language System for Spoken Language Applications.* Computational Linguistics, 18:1 (Marzo 1992), págs. 20-40.

[9] Sells Peter et al. *Fundational Issues in Natural-Language Processing.* MIT Press, 1991.

[10] Smith George. *Computers and Human Language.* Oxford University Press, 1993.

[11] Winograd, Terry. *Language as a Cognitive Process,* Volume 1: Syntax. Addison-Wesley, 1983.