JACC un entorno de generación de Procesadores de Lenguajes

Jorge Aguirre, Valentina Grinspan, Marcelo Arroyo, Jorge Felippa, Guillermo Gomez[1]

Resumen

En el presente trabajo se exponen los puntos más relevantes del diseño e implementación de jacc, un entorno de generación de procesadores de lenguajes o de compilación de compiladores, basado en un formalismo que brinda la posibilidad de utilizar las posibilidades de evaluación de las gramáticas de atributos, conjuntamente con la potencia de los esquemas de traducción. La primera versión genera analizadores sintácticos LALR y evaluadores concurrentes de atributos. El diseño se ha realizado dentro la Tecnología de Orientación a Objetos y se ha usado Java como lenguaje de implementación. El lenguaje especificación que brinda jacc sigue el estilo yacc, que prácticamente es un estándar. También se describen las extensiones que se harán a la versión inicial para dotarla de la posibilidad de usar otros métodos de análisis, mejorar la eficiencia y dotarlo de un módulo de generación de código móvil seguro.

Palabras claves: gramáticas de atributos, esquemas de traducción, compilador de compiladores, concurrencia, análisis sintáctico, análisis lexicográfico, Orientación a objetos.

Agradecimientos: Queremos expresar nuestro agradecimiento a Jorge Guazzone y Dario Nicolino que desarrollaron el módulo de generación de analizadores sintácticos LALR y a Daniel Romero que desarrollo el generador de analizadores lexicográficos.

1. Introducción

1.1. Antecedentes

La especificación de lenguajes formales y el desarrollo de sus correspondientes procesadores, compiladores, intérpretes o interfaces con el usuario y posteriormente la construcción de herramientas que los generan, ha ocupado un lugar preponderante en las Ciencias de la Computación, con importantes consecuencias sobre la industria de software [Aho72][Tra85][Aho86][App98][Fra95][Wai84][Wai92]. Pese a los grandes avances realizados en el área, aún quedan aspectos importantes por resolver; a su vez el avance tecnológico incorpora nuevas problemáticas a su campo de acción.

Los modelos teóricos que sustentan los lenguajes de especificación usados actualmente son las gramáticas de atributos [Pag81][Wai84] y los esquemas de traducción [Aho86]. Las herramientas más usadas yacc [Joh75] - originalmente desarrollada dentro del entorno de desarrollo de Unix y luego migrada a prácticamente todas las plataformas - y actualmente Javacc [Javac], brindan un mecanismo que resulta de una débil integración de los modelos mencionados. Esta integración permite trabajar con un conjunto restringido de esquemas de traducción - sin modificación del orden de los símbolos no terminales en la traducción -, y agrega la facilidad de usar atributos asociados a los símbolos, que son evaluados en un orden prefijado, aquel en el que se ejecutan las acciones en que son evaluados. Este modelo sólo permite resolver problemas que puedan expresarse con Gramáticas de atributos a izquierda *L-attributed grammars* , no pudiendo ser expresados aquellos que escapen a este tipo, tales como el de verificación del tipo de una expresión Ada [Aho86], en estos casos el usuario debe recurrir a la generación de un árbol sintáctico, usando las facilidades brindadas por la herramienta, y luego a construir su propio evaluador, que implemente las estrategias de recorrido necesarias. Por otra parte si usa yacc que está basado en un analizador ascendente - LALR(1) - la herencia debe ser emulada por el usuario, lo que en muchos casos conduce a que la gramática deje de ser LALR(1), no pudiendo ser tratada.

[1] Universidad Nacional de Río Cuarto. e-mails: jaguirre@dc.uba.ar, vgrinspan@pragma.com.ar, marroyo@dc.exa.unrc.edu.ar, jfelippa@dc.exa.unrc.edu.ar, ggomez@dc.exa.unrc.edu.ar

Este trabajado ha sido realizado en el marco de proyectos subsidiados por la SECyT de la UNRC y por la Agencia Córdoba Ciencia.

En este trabajo se describe el desarrollo de un entorno de generación de procesadores de lenguajes - **Jacc -** basado en un nuevo modelo, que integra de una manera más general a los Esquemas de Traducción con las gramáticas de Atributos. El nuevo formalismo ha sido llamado Esquemas de Traducción con atributos, *Attibute Translation Schems* (ATS).

Las gramáticas de atributos (AG) permiten expresar propiedades dependientes de contexto, asociando atributos - variables tipadas - a los símbolos de una gramática y reglas de evaluación de atributos a las producciones. Las reglas de evaluación deben tener la forma $a_0 = f(a_1, a_2, a_3,..., a_n)$ - donde los a_i deben ser atributos de los símbolos de la producción, y *f* una aplicación estrictamente funcional -. Para la ejecución de estas reglas no se determina un orden explícito de evaluación sino que ellas expresan un conjunto de ecuaciones simultáneas que pueden evaluarse en cualquier orden que respete la dependencia implícita entre los valores de los atributos.

Las implementaciones de los esquemas de traducción, *Trasnslation Schems* (TS), en cambio, agregan acciones intercaladas entre los símbolos de la parte derecha de las producciones de una gramática; su semántica consiste en ejecutar tales acciones de izquierda a derecha en el orden en que aparecen, como hojas, en los árboles de derivación. En este caso el orden de ejecución es parte esencial de la gramática, ya que estas acciones pueden producir efectos colaterales, dejando, por lo tanto, de valer el principio de transparencia referencial que regía en evaluación de las AGs.

Yacc, Javacc y otros generadores de procesadores de lenguaje actuales utilizan TSs enriquecidos con atributos. Permiten hacer referencia a atributos dentro de las acciones de traducción, pero el computo de atributos se realiza mediante acciones. El orden de evaluación de los atributos, queda así determinado, por el orden en que estas deben ejecutarse. Este mecanismo pierde generalidad respecto a las AGs puesto que impone un orden de evaluación que compatible sólo con las llamadas Gramáticas de atributos por izquierda, restricción que se extiende a la utilización de atributos dentro de las acciones de traducción**.**

1.2. Formalismo propuesto, esquemas de traducción con atributos (ATS)

En un ATS se declaran separadamente las acciones de traducción y las reglas de evaluación de atributos. Las acciones deben ser ejecutadas en el mismo orden relativo que en los TSs. Las reglas de evaluación de atributos, como en las AGs, pueden ejecutarse en cualquier orden relativo, que respete la dependencia entre los atributos. Finalmente para que una acción sea ejecutada, es necesario que se hayan computado todos los atributos de los que hace uso.

La implementación que hace Jacc de un ATS consiste en un conjunto de procesos concurrentes que se sincronizan entre sí para lograr un orden de evaluación viable. Esto se verá esquemáticamente mas adelante y ha sido descripto con más detalle en un trabajo ya publicado sobre el entorno de ejecución de los traductores generados por jacc [Agu98] .

Jacc ha sido diseñado usando la tecnología de Orientación a Objetos, como así también el lenguaje especificación que debe utilizar el usuario. El diseño ha sido realizado siguiendo el proceso unificado de desarrollo de software - UML - [Han98], siguiendo diversos patrones de diseño [Gam95]. Jacc soporta las facilidades de las Gramáticas de atributos condicionales [Boy96].

Se han desarrollado distintas extensiones para incorporar a jacc: un módulo de análisis lexicográfico que permite especificaciones basadas en Expresiones Regulares Traductoras [Agu99], un módulo de análisis descendente no recursivo basado en un algoritmo que emula análisis ascendente bajo control descendente [Agu98b] y un método de evaluación paralela de GA de tipo NC(1) con mínima comunicación entre procesos [Arr00]. Se han iniciado trabajos sobre Código móvil seguro

Proof Carrying Code (PCC) [Nec96] [App99] [App00] con vistas a la incorporación de un módulo de generación de código de este tipo.

2. El entorno de generación de procesadores de lenguajes Jacc

2.1. Esquema general de funcionamiento

Jacc permite generar un procesador de lenguajes, traductor o compilador, a partir de su especificación por medio de un ATS y de la especificación del analizador léxico de su lenguaje de entrada, como se muestra en la figura 1.

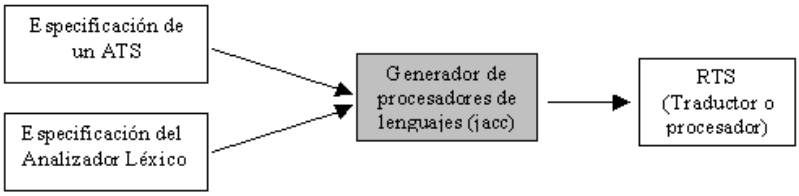

Figura 1. Entrada y salida del generador de procesadores de lenguajes.

El generador de procesadores de lenguajes es, en esencia, un compilador que a partir de las especificaciones de entrada obtiene un procesador del lenguaje especificado. Este procesador de lenguaje podrá ser, por ejemplo, un compilador, que produzca programas ejecutables a partir de programas fuentes, o un interprete, una interface de usuario, que permita interactuar con un sistema de información mediante un lenguaje ad hoc. La salida del generador provee el sistema de tiempo de ejecución que ejecuta al ATS especificado en la entrada, denominado RTS *Run Time System*

2.2. Lenguaje de especificación de un ATS

Considerando la amplia difusión del par de generadores - de analizadores lexicográfico y sintáctico- Lex y Yacc [Lev92], se ha adoptado el mismo estilo para los lenguajes de especificación

La figura 2 muestra la gramática del lenguaje de especificación, la misma se divide en tres partes: la primera alberga declaraciones Java que el usuario incluye en su especificación, en ella se pueden importar paquetes o definir clases propias. En la segunda parte se incluyen las declaraciones propias de la gramática, como la declaración de los atributos, de los símbolos terminales y no terminales y las declaraciones de precedencia y asociatividad. Finalmente, la tercera parte alberga las reglas del ATS, cada una de las cuales puede tener asociado un conjunto de reglas de evaluación de atributos, una condición y acciones de traducción imbuidas en la parte derecha de la producción. Las reglas de evaluación se deben escribir despues de la producción precedidas de la cláusula ATTRIBUTION, la condición también debe figurar despues de la producción, precedida de la cláusula CONDITION; finalmente las acciones de traducción son código java encerrado por llaves. Estas pueden intercalarse entre los símbolos de la parte derecha de una producción. El conjunto de atributos de un símbolo debe ser declarado como una clase y cada uno de ellos como una variable miembro.

2.3. Lenguaje de especificación del analizador léxico.

Para especificar los componentes léxicos se usa un lenguaje patrón-acción, similar al utilizado por la herramienta Lex, que usa a Java como lenguaje huésped. La especificación del usuario debe retornar el token correspondiente y computar sus atributos comunicándolos mediante una variable global destinada a este fin.

2.4. Lenguaje de implementación.

Se eligió a Java [Gos96] como lenguaje de implementación del generador, tanto como lenguaje huésped para la definición de las reglas de atribución y acciones de traducción y como lenguaje de implementación de los traductores generados. Esta elección se basó en que: 1) se deseaba usar la tecnología de OO para el entorno; 2) se requería que el lenguaje soportara concurrencia y manejo de excepciones; 3) se deseaba que la implementación fuera multiplataforma.

2.5. Método de parsing usado por los traductores generados

Se optó para la implementación por un método ascendente de parsing, dentro de esta familia se eligió al método LALR puesto que constituye un punto intermedio, de aceptable generalidad entre el bajo costo y el alto poder de los métodos SLR Y LR.

ATS → DecJava DecGram %% Gram
DecJava → % { NoNameJavaPackage %}
DecJava → λ
NoNameJavaPackage → Σ^+

DecGram → JavaClassDefinition DG
DecGram → DG
JavaClassDefinition → Σ^+

DG → % **token** < Ident > TokenIdentList
DG → % **token** TokenIdentList
DG → % **type** < Ident > SymbolIdentList
DG → % **token** SymbolIdentList
DG → % **right** IdentList
DG → % **left** IdentList
DG → % **start** IdentSymbol
DG → λ

Gram → Rule
Gram → Rule ; Gram
Rule → Head : ManyBodys
Head → IdentSymbol
ManyBodys → Body
ManyBodysn → Body | ManyBodys

Body → LocalDec TransRule Attribution Condition

LocalDec → **LOCAL** { JavaVariableDeclarations }
LocalDec → λ
JavaVariableDeclarations → Σ^+

TransRule → Ident TransRule
TransRule → { JavaCode } TransRule
TransRule → λ
JavaCode → Σ^+

Attribution → **ATTRIBUTION** { JavaAssignationList }
Attribution → λ
JavaAssignationList → Σ^+
Condition → **CONDITION** [JavaLogicalExpression]
Condition → **CONDITION** [JavaLogicalExpression]
IF FALSE { JavaCode }
Condition → λ
JavaLogicalExpression → Σ^+

TokenIdentList → IdentToken
TokenIdentList → IdentToken , TokenIdentList

TokenIdent → (**A** | ... | **Z**) (**A** | ... | **Z** | **0** | ... | **9** | _)*

SymbolIdentList → IdentSymbol
SymbolIdentList → IdentSymbol , SymbolIdentList

SymbolIdent → (**a** | ... | **z**) (**a** | ... | **z** | **A** | ... | **Z** | **0** | ... | **9** | _)*

IdentList → Ident
IdentList → Ident , IdentList
Ident → IdentToken
Ident → IdentSymbol

Figura 2. Gramática del lenguaje de especificación.

3. Descripción general del modelo de implementación

El diseño del sistema de tiempo de ejecución - RTS - independiza el proceso de análisis sintáctico del proceso de evaluación de atributos y del de ejecución de acciones de traducción.

El resultado es un modelo de implementación de ATS's que consiste de un analizador sintáctico de la gramática subyacente, extendido con un conjunto de procesos destinados a la evaluación de

atributos y a la ejecución de las acciones de traducción, que se ejecutan concurrentemente, y de un conjunto de objetos compartidos a través de los cuales se comunican tales procesos.

Las ocurrencias de los atributos son objetos compartidos que crea el sistema. La clase de una ocurrencia cuenta con los métodos *put* y *get* destinados a definir o usar su valor. Estos métodos sirven para sincronizar el acceso de los procesos de evaluación a una ocurrencia de un atributo. *get* entrega el valor de la ocurrencia o duerme al proceso que lo invoca si el valor aun no ha sido asignado, mientras que *put* asigna un valor y despierta a los procesos que esperan por él. Este mecanismo es manejado por el sistema en forma transparente, como se muestra en la figura 9, donde los atributos son referidos por su nombre.

Los procesos que ejecuta el modelo pueden ser clasificados en tres tipos:

1. El parser, proceso secuencial que a lo largo del análisis sintáctico inicializa procesos de los otros dos tipos;
2. Los procesos de traducción, asociados a las acciones de traducción iniciados por el sistema, en la secuencia en que deben ser ejecutados. Estos procesos corren en concurrencia con el analizador sintáctico y los procesos del tercer tipo.
3. Los procesos de evaluación de atributos asociados a las reglas de evaluación, iniciados por el sistema y ejecutados concurrentemente con todos los otros, incluso con los del mismo tipo, con la única restricción de la dependencia entre atributos.

Los procesos de tipo 3 se comunican entre sí y hacia los procesos de tipo 2 mediante los mencionados objetos compartidos en donde se alojan los valores de los atributos. Cuando un proceso hace referencia a un atributo aún no evaluado, se bloquea hasta que el valor esté disponible.

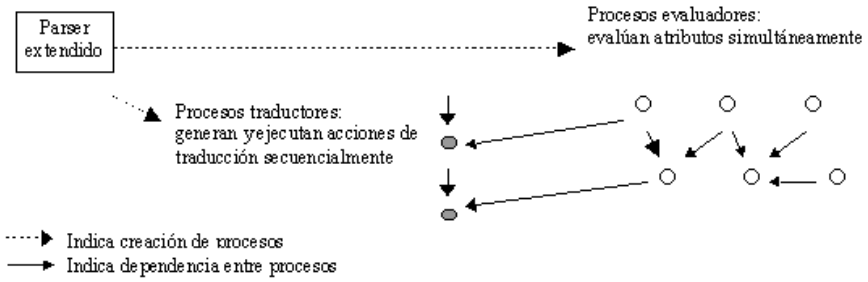

Figura 3. Implementación de un ATS: componentes y su interrelación.

4. Generador de procesadores de lenguajes (jacc)

El generador de procesadores de lenguajes es una herramienta que dado un ATS genera automáticamente un traductor capaz de analizar sintáctica y semánticamente las cadenas que el ATS especifica.

El generador está estructurado en forma similar a la de un compilador. Sus módulos principal son: un módulo de análisis lexicográfico, un módulo de análisis sintáctico, un módulo de análisis semántico y otro de generación de código.

En la figura 4 se presenta el diseño de módulos del generador, se utiliza la notación gráfica GDN ("Graphical Design Notation") [Ghe91] para mostrar esquemáticamente los módulos del sistema y sus relaciones.

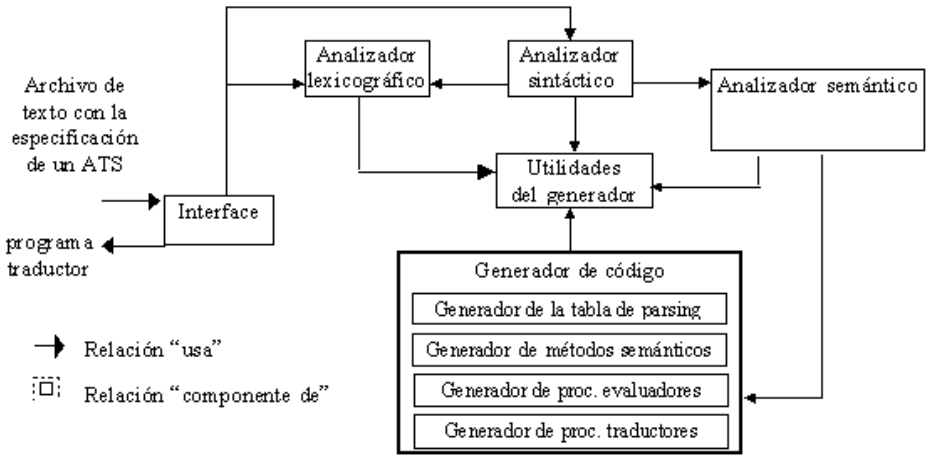

Figura 4. Módulos del generador de traductores

Interface. Toma el requerimiento del usuario para la generación de un traductor y retorna el traductor generado o eventualmente el reporte de errores cometidos. Recibe un archivo de texto con la especificación de un ATS e inicializa a los analizadores lexicográfico y sintáctico. La salida producida por una compilación exitosa es un programa Java que implementa el traductor.

Analizador lexicográfico. Lee caracteres del archivo de texto que contiene la especificación de la gramática, reconoce lexemas y retorna tokens. Es utilizado por el módulo de interface que lo inicializa indicándole el archivo de lectura, y por el analizador sintáctico el cual hace los requerimientos de tokens.

Analizador sintáctico. Recibe el requerimiento del módulo de interface de analizar el archivo de entrada. Utiliza el modulo de análisis lexicográfico, del cual obtiene los tokens del archivo de entrada, y a medida que realiza el análisis de la cadena de entrada, invoca al analizador semántico.

Analizador semántico. implementa diversas verificaciones estáticas sobre la validez del ATS de entrada.

Generador de código. Este módulo es responsable de construir la tabla de parsing del traductor de salida, de generar el código Java de los procesos traductores y evaluadores, y de las otras componentes del traductor de salida que dependan del ATS de entrada. Finalmente debe integrar estos segmentos de código para conformar el traductor generado. El generador de código utiliza servicios del módulo de utilidades

Para construir la tabla de parsing LALR(1) se construye directamente el conjunto canónico de conjuntos de items LALR(1), sin la construcción previa del LR(1). Como la representación matricial de las tablas conduce a matrices ralas, se utiliza una representación que aprovecha esta característica para economizar espacio.

Utilidades del generador. En este módulo se agrupan todas las utilidades requeridas por los otros módulos.

4.1. Clases del generador

El diseño de las clases del generador surge del anterior diseño de módulos. De esta manera se obtienen los grupos de clases: clases de análisis lexicográfico y sintáctico, clases de análisis semántico, clases de generación código, clases de interface y clases de utilidades.

4.2. De análisis lexicográfico y sintáctico

En el diseño hay dos pares de analizadores, léxico y sintáctico. Un par de ellos actúa en tiempo de generación, ocupándose del parsing de la especificación de un ATS, mientras que el otro par está destinado a actuar en tiempo de ejecución del traductor generado. Consiguientemente se definen cuatro clases: 'LexicalATS' y 'ParserATS' que implementan el primer par de analizadores, y 'LexicalLex' y 'ParserLex' que implementan al segundo par.

Para implementar el primer par de las clases mencionadas se utilizo la herramienta Javacc.

4.3. De análisis estático

La responsabilidad de estas clases consiste en analizar la validez del uso de atributos en las reglas de evaluación y traducción del ATS especificado y en las acciones asociadas a las expresiones regulares de la especificación del analizador lexicográfico. Por ejemplo, se controla que no se asocie a un símbolo un atributo que no posee, o que se utilice un atributo no declarado.

La representación interna de una regla del ATS es un objeto de la clase 'Rule', y una regla del analizador lexicográfico es un objeto de la clase 'RegExp', como se muestra en la figura 5.

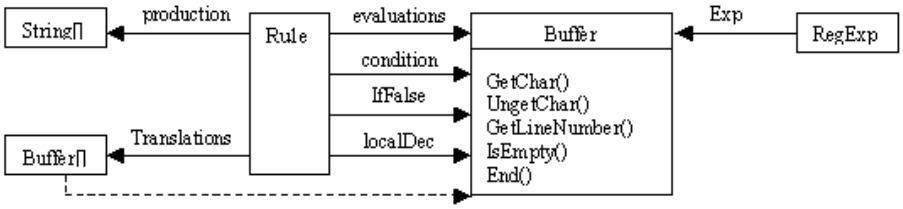

Figura 5. Representación interna de reglas de ATS y de un analizador lexicográfico

Un objeto de la clase 'Rule' posee un arreglo de strings que indican el tipo de cada símbolo de la producción, una lista de buffers, cada uno de los cuales posee el código de una acción de traducción, un buffer con las declaraciones locales, un buffer con el código de las reglas evaluación, un buffer con el código de la condición y otro con el código que debe ser ejecutado cuando la condición es falsa.

Un objeto de la clase 'RegExp', simplemente posee un buffer que contiene el código asociado a la expresión regular.

Para realizar las verificaciones mencionadas se define una clase 'ScanRule' cuya función es reconocer los códigos contenidos en un objeto 'Rule', y una clase 'ScanExp' responsable de controlar el código contenido en el buffer de un objeto 'RegExp'. Debido a que sus métodos deben ejecutarse concurrentemente, son subclases de la clase Java 'Thread'.

La comunicación entre las clases de análisis sintáctico - responsables de analizar la entrada y crear los objetos de 'Rule' y 'RegExp' - y las clases de reconocimiento, se establece a través de una clase 'Scanner' que actúa de interface entre ellas. Esta clase provee métodos de acceso a 'ScanRule' y 'ScanExp' conteniendo además los parámetros generales de reconocimiento, como una tabla de los tokens definidos en la gramática y una tabla de los tipos de símbolos de la gramática.

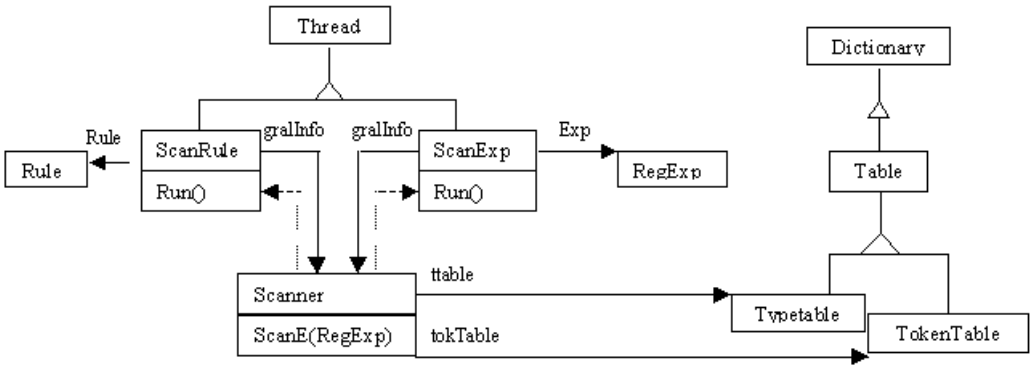

Figura 6. Clases de reconocimiento.

4.4. De generación de código

Estas clases - figura 7 - son responsables de generar código especifico o de integrar código ya generado en la definición de una clase. La salida de estas clases es almacenada en archivos de nombre estándar de manera tal que la comunicación entre una clase que genera un determinado código y otra que lo utiliza se establece a través de los archivos creados.

Por otra parte, la generación de código podría iniciarse cuando aun otras acciones de análisis sintáctico o semántico están pendientes, por lo tanto, es conveniente que se permita su ejecución concurrente con otras acciones del generador. En consecuencia las clases de generación de código son subclases de la clase de Java 'Thread'.

Los elementos comunes a la generación de código son encapsulados en la clase 'CodeGenerator', y las demás clases son extensiones de ella. Esta clase provee métodos que permite el manejo de archivos.

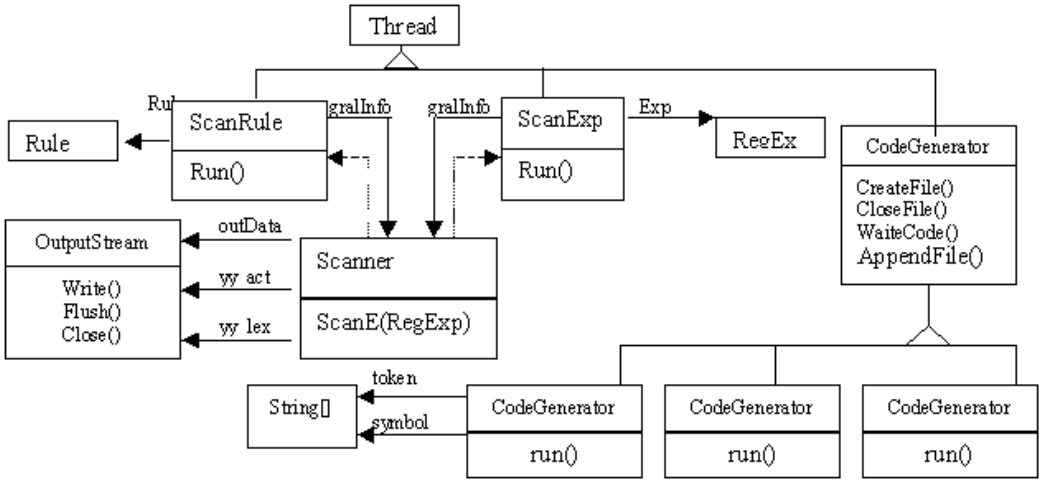

Figura 7. Clases de generación de código

4.5. De interface

La interface de nivel superior del generador es provista por una clase denominada 'Jacc' (por Java compiler of compilers). Esta clase define una interface uniforme para la funcionalidad del generador y actúa como fachada del subsistema, ya que ofrece a los usuarios una interface simple y fácil de utilizar.

Los usuarios de la herramienta no tienen acceso a los objetos del subsistema directamente, sino que se comunican con él enviando requerimientos a ‘Jacc’, quien dirige los mismos a los objetos correspondientes del generador.

La clase ‘Jacc’ es responsable de inicializar la generación de un traductor: debe recibir los archivos en donde se encuentran la especificación del ATS y la especificación del analizador lexicográfico. Además, la clase ‘Jacc’ es responsable de crear objetos de la clase ‘Parser’ y ‘Lexical’ e iniciar la generación del traductor invocando al método parse().

En la siguiente figura se muestra el esquema de todas las clases del generador.

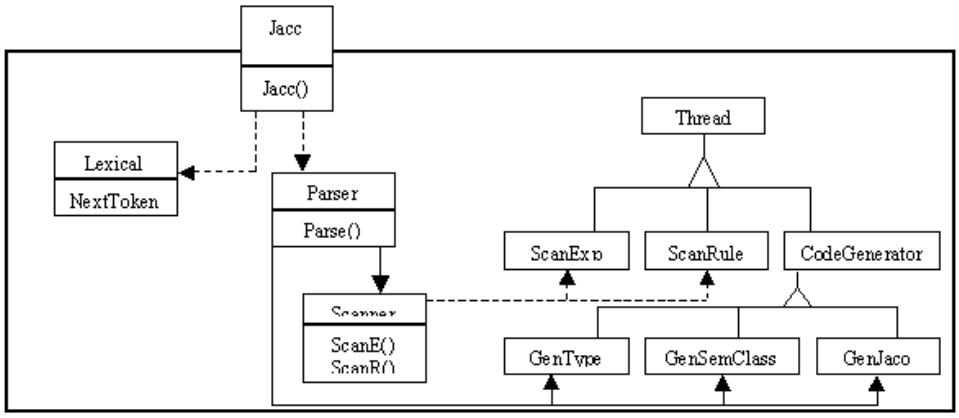

Figura 8. Clases del generador de procesadores y su interface

5. Utilización del generador

Para generar el traductor definido por un esquema de traducción con atributos (ATS), el usuario del generador debe invocar a este último escribiendo:

jacc [-opciones] archivo1 [archivo2] donde

- *archivo1* es un archivo de texto de extensión "jacc" con la especificación del ATS, la cual debe respeta la sintaxis definida en la figura 2;
- *archivo2* es opcional y se especifica si se desea realizar análisis lexicográfico. Es un archivo de texto de extensión "lex" con la especificación del conjunto de expresiones regulares a reconocer. Si el usuario omite el archivo2 en la invocación al generador, el traductor generado tomará un carácter por vez de la cadena de entrada;
- las opciones que pueden incluirse son:

  -*nowrite* al incluirse esta opción el generador solo analiza semánticamente el ATS de entrada pero no genera código para el traductor.

  -*noname* al incluirse esta opción el generador asigna a los símbolos sin atributos declarados un atributo entero por defecto, el cual puede ser referenciado a través del nombre del símbolo, es decir, no hace falta dar un nombre el atributo;

Por defecto se ejecuta jacc generando código y creando únicamente los atributos declarados. Los archivos "lex" y "jacc" deben estar ubicados en el mismo directorio que la clase "jacc.java".

Si no ocurre ningún error y no se utilizó la opción *-nowrite*, el generador crea las clases que implementan al traductor del ATS de entrada. De estas clases la principal es la clase de interface que permite acceder a todas las clases del traductor. El nombre de esta clase es el mismo que el del

archivo jacc de entrada pero con extensión "java". Para obtener un programa ejecutable el usuario debe compilar dicha clase utilizando el compilador de java.

5.1. Un Ejemplo de la especificación de un ATS

El siguiente ejemplo muestra la utilización de un ATS para especificar un esquema de compilación de expresiones en un sistema de tipos similar al de ADA, todas las operaciones deben realizarse en tipo superior de toda la expresión[2]. Por brevedad el ATS mostrado sólo imprime la secuencia de operaciones a realizar

```
%{
%}

class Expresion{
  String tipo;
  String gral;
}

%token ENTERO, REAL
%token MAS
%type <Expresion> E

%start S

%%
S : E
      ATTRIBUTION
            {$1.gral = $1.tipo;}
 ;

E : E
   MAS       {if ($$.gral == "Entero")
                  System.out.println("<+E>");
            else
                  System.out.println("<+R>");}
   E

      ATTRIBUTION

      {$$.tipo = (($1.tipo == "Entero" && $3.tipo == "Entero") ? "Entero": "Real")

            $1.gral = $$.gral;
            $3.gral = $$.gral;
           }
 |ENTERO
      ATTRIBUTION
            {$$.tipo = "Entero";}
 |REAL
      ATTRIBUTION
            {$$.tipo = "Real";}
 ;
```

Figura 9. Ejemplo de ATS

6. Conclusiones y trabajos futuros

Se ha obtenido una primer versión de jacc. El entorno desvincula al usuario de la consideración de todo control sobre el proceso de evaluación. El lenguaje de jacc se integra el formalismo usado al

[2] Obsérvese que la gramática no es *L_attributed*, dado que la primera regla de atribución computa un atributo heredado en función de un atributo del mismo signo, por tanto está fuera del alcance de yacc y Javacc

paradigma de OO y resulta totalmente familiar para los usarios de yacc. El entorno puede ejecutar sobre cualquier plataforma que ejecute Java.

Se planea incorporar las siguientes extensiones:

• Incorporación de un módulo alternativo de generación de analizadores lexicográficos basado en Expresiones Regulares Traductoras Lineales. Este formalismo [Agu99] permite la especificación conjunta de sintaxis y semántica de los componentes léxicos, siguiendo el mismo estilo de los esquemas de traducción.

• Incorporación de un generador de analizadores sintácticos descendentes LL(1). Se encuentra en desarrollo este módulo que permitirá elegir el generador de analizadores sintácticos que usara el entorno. Permitirá evaluación concurrente como en la versión actual, o usando el método de evaluación *down-up* [Agu98b]

• Incorporación de un método alternativo de evaluación concurrente de atributos [Jou91] de máximo paralelismo y mínima comunicación basado en un algoritmo ya desarrollado por los autores [Arr00] para gramáticas NC(1) [Wuu97][Wuu99].

• Incorporación de un módulo de generación de código móvil seguro. Con la gran difusión del uso de código móvil que ha sido recibido de otro sistema, la protección de la integridad del receptor de código invasivo ha cobrado enorme importancia. Un enfoque que ha generado una activa y creciente línea de investigación es el *Proof Carrying Code* [App00][App99][Wai99][Dea99] que fuera introducida por Necula [Nec96]. Está técnica se basa en la generación conjunta del código ejecutable y una prueba de su seguridad, que el receptor pueda correr antes de ejecutar el código . El grupo ha comenzado a trabajar en esta línea, la conclusión deseable de la investigación sería la inclusión en el entorno del módulo mencionado.

Referencias bibliográficas

[Agu98a] J. Aguirre, V. Grinspan, Marcelo Arroyo, “Diseño e Implementación de un Entorno de Ejecución, Concurrente y Orientado a Objetos, generado por un Compilador de Compiladores”, Anales ASOO 98 JAIIO, pp. 187-195, Buenos Aires1998.

[Agu98b] J. Aguirre, V. Grinspan, Marcelo Arroyo, “Un Parser Top Down que Emula el Comportamiento Bottom up”, Anales WAIT 98 JAIIO, pp 45-57, 1998.

[Agu99] J. Aguirre, G. Maidana, M. Arroyo, “Incorporando traducción a las Expresiones Regulares”, Anales del WAIT 99 JAIIO, pp 139-154, 1999.

[Aho72] A. V. Aho, J. D. Ullman, “The Theory of Parsing, Translation, and Compiling”, Prentice-Hall, INC., Englewood Cliffs, New Jersey. 19

[Aho86] A.V. Aho, R. Sethi, J.D. Ullman, “Compilers: Principles, Techniques, and Tools”, Addison Wesley, 1986.

[App00] A. W. Appel, A. P. Felty, “A Semantic Model of Types and Machine Instructions for Proof-Carrying Code”, 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.(POPL '00), pp. 243-253, January 2000.

[App98] A. W. Appel, “Modern Compiler Implementation in Java”. Cambridge University Press, ISBN: 0-521-58388-8, 1998.

[App99] A. W. Appel, E. W. Felten, “Proof-Carrying Authentication”. 6th ACM Conference on Computer and Communications Security, November 1999.

[Arr00] M. Arroyo, N. Florio, J. Aguirre, “Un generador de evaluadores de gramáticas de atributos NC(1) de máximo paralelismo sin sincronización entre procesos”, UNRC, Anales del CACIC 2000 pp. 523-526, Ushuaia 2000.

[Boy96] J. T. Boyland, “Conditional Atribute Grammars”, ACM Transactions on Programming Languages and Systems, Vol. 18, No. 1, January 1996, pages 73-108.

[Dea99] R. D. Dean, “Formal Aspects of Mobile Code Security”, PhD thesis, Princeton University, January 1999.

[Fra95] C. Fraser, “Retargetable C Compiler : Design and Implementation”, Benjamin Cummings Publishing Company, 1995

[Gam 95] Gamma , E. et al. “Design Patterns. Elements of Reusable Object Oriented Software”. Addison Wesley.

[Ghe91] Ghezzi,C. et al. “Ingegneria del Software. Progettazione, sviluppo e verifica”. Mondadori Informatica S.p.A., Milano.

[Gos96] J. Gosling, B. Joy, Steele, “The Java Lenguage Specification”, Addison Wesley, 1996.

[Han98] E. Hans-Erik, M. Penker, “UML Toolkit”, John Wiley & Sons, Inc, 1998.

[Hol90] A. Holub, “Compiler Design in C”, Perintice Hall, 1990.

[Javac] http://falconet.inria.fr/~java/tools/JavaCC_0.6/doc/DOC/index.html

[Joh75] Johnson, S.C. “Yacc - yet another compiler compiler”. Computing Science Technical Report. AT&T.

[Jou91] M. Jourdan. “A Survey of Parallel Attribute Evaluation Methods. Attribute Grammars, Applications and Systems” – Lecture Notes in Computer Science, Springer-Verlag. Pag: 234-255, vol. 545. 1991.

[Lev92] J. Levine, T. Mason, D. Brown. “Lex & Yacc”.O’Reilly & Associates, Inc. 1992

[Nec96] G. Necula, P. Lee. “Safe Kernel Extensions Without Run-Time Checking”. Second Symposium on Operating Systems Design and Implementation. Seatle. 1996.

[Pag81] F. G. Pagan, “Formal Specification of Programming Languages, a panoramic primer”. Prentice-Hall, INC., Englewood Cliffs, New Yersey, 1981.

[Tra85] J. Trambley, P. Sorenson, “The Theory and Practice of Compilers Writing”. Mc Graw Hill, 1985.

[Wai84] Waite, Goos. “Compiler Construction”. Springer-Verlag – 1984.

[Wai92] Waite, Carter. “An Introduction to Compiler Construction”. HaperCollins College Publishers. ISBN: 0-673-39822-6. 1992.

[Wai99] D. S. Wallach, “A New Approach to Mobile Code Security”, PhD thesis, Princeton University, January 1999.

[Wuu97] Y. Wuu, “A Classification of non-circular attribute grammars”, Research Report, Computer and Information Science Dept., National Chiao-Tung Univ., Hsinchu, Taiwan, 1997.

[Wuu99] Y. Wuu, “A Finest Partitioning Algorithm for Attribute Grammars”, March 1999, Second Workshop on Attribute Grammars and their Applications – WAGA99.