Generador de Evaluadores Estáticos para Gramática de Atributos Multiplanes

magGen

Arroyo, Marcelo Daniel; Kilmurray, Gerardo Luis; Picco, Gonzalo Martín

Departamento de Computación,
Facultad de Ciencias Exactas, Físico-Químicas y Naturales,
Universidad Nacional de Río Cuarto - http://www.unrc.edu.ar/
marroyo | gkilmurray | gpicco@dc.unrc.edu.ar.

Resumen Las **Gramática de Atributos** (GA) son un formalismo que poseen el poder descriptivo de las Gramáticas Libres de Contexto (CFG) y la expresividad de los lenguajes funcionales, para definir la semántica de un lenguaje.
En 1998, Wuu Yang caracteriza una nueva familia de GA, denominada **Gramática de Atributos Multiplanes** (MAG), que permiten una mayor expresividad, presentando un algoritmo para su evaluación, basado en secuencias de visita, en tiempo polinomial.
La familia MAG, también denominada NC(1), pertenece a las GA No Circulares que pueden ser computadas con planes de evaluación estáticos.
magGen es una herramienta que genera, de manera automática, evaluadores estáticos para la familia MAG, basada en la teoría presentada por Wuu Yang. La salida de **magGen** es un conjunto de módulos C++ optimizados para su directa utilización o integración con otras herramientas.

1. Introducción

Desde que D. Knuth introdujo en 1966 las Gramáticas de Atributos (GA)[7], estas se han utilizado ampliamente para el desarrollo de herramientas de procesamiento de lenguajes formales, como compiladores, intérpretes, traductores, así como también para especificar la semántica de lenguajes de programación.

magGen es el aporte principal de este trabajo, su denominación esta dada por la combinación de las sílabas **mag**, abreviatura de *Multi-plans Attribute Grammar* y **Gen** por *generator*. Agregando el hecho de que los evaluadores son estáticos, se obtiene un significado completo como "Generador de Evaluadores Estáticos para MAG".

1.1. ¿Qué es magGen, qué hace y qué lo distingue de los demás?

magGen es una herramienta generadora de evaluadores estáticos para la familia MAG. Hasta el momento no existen herramientas basadas en esta familia de GA.

magGen se encuadra dentro de un conjunto de herramientas complementarias, como generadores de parsers, con el objetivo de desarrollar un framework para el procesamiento de lenguajes basado en especificaciones.

La importancia que adquiere **magGen**, además de basarse en la familia MAG, radica en que genera la mínima cantidad de planes necesarios y plausibles para evaluar la gramática, y en que el evaluador generado es *estático*, lo que garantiza que no hay procesamiento "overhead" durante la evaluación, sino que el mismo se realizó en tiempo de generación y compilación del evaluador propiamente.

2. Preliminares

En una **Gramática de Atributos** (GA), se relaciona cada símbolo de una *Gramática Libre de Contexto* con un conjunto de atributos. Cada regla o producción tiene asociado un *conjunto de reglas semánticas*, denominadas también *ecuaciones*, que establecen la forma de asignación a *atributos* de valores denotados por la aplicación de una función.

Un evaluador de gramáticas de atributos debe seguir un orden de evaluación consistente, con respecto a las dependencias entre los atributos de una gramática de atributos, el cual es una secuencia (orden parcial) de instancias de atributos con la siguiente restricción: Dada una regla $r_j^p : X_0.a_0 = f(\ldots, X_i.a_i, \ldots)$ en una producción p,$X_i.a_i$ deberá preceder a $X_0.a_0$.

Una gramática de atributos GA es "**Circular**" si y sólo si existe un Árbol Sintáctico Atribuido $T(GA)$, tal que su grafo de dependencias $GD(T)$ contiene al menos un ciclo.

Si una GA contiene dependencias circulares no es posible encontrarle un orden de evaluación consistente. Esto se conoce como "*Problema de la circularidad*", el cual se ha demostrado ser intrínsecamente exponencial[5]. El propósito de muchas investigaciones, ha sido descubrir nuevas familias o subgrupos de GAs, para las cuales puedan detectarse circularidades con algoritmos de complejidad menor.

En 1998, Wuu Yang en [1] caracteriza una nueva familia dentro de las WGAs[1], denominada **Gramática de Atributos Multiplanes**[2], presentando un algoritmo de evaluación basado en secuencias e visita en tiempo polinomial en el numero de símbolos y producciones.

Los métodos de evaluación estáticos deben tener en cuenta todos los árboles sintácticos posibles a ser generados por la gramática y calcular las dependencias, entre las instancias de los atributos, para cada uno de ellos.

Un árbol sintáctico se construye a partir de la aplicación sucesiva de producciones de la gramática. Una instancia de una producción en un árbol sintáctico tiene como *contexto inferior* a las instancias de las producciones aplicadas a los símbolos no terminales de la parte derecha.

[1] Gramáticas Bien Definidas.

[2] La familia de las MAG contiene a la familia de las OAG presentadas por Uwe Kastens[4] en 1980.

Dada una producción p se deben considerar tres tipos de dependencias que definen el contexto de la misma: Directas obtenidas por las ecuaciones de p, Impuestas por el contexto superior e Impuestas en el contexto inferior.

Instancias diferentes de una producción p tendrán las mismas dependencias directas, pero podrán tener diferentes dependencias impuestas por los contextos inferiores y superiores. Es necesario entonces, determinar todas las *dependencias posibles* entre las instancias de los atributos que ocurren en una producción para luego poder determinar todas los *planes de evaluación posibles* entre los atributos que ocurren en una producción.

El concepto de secuencias de visita, desde el punto de vista en el cual lo presenta Kastens, es aplicable no sólo a las OAG sino también para WDAGs. Dado un nodo de un árbol atribuido T; una secuencia de visita, en el nodo, es una secuencia compuesta por tres operaciones o acciones:

visit(child,i) indica que el evaluador debe moverse (visitar) el nodo hijo *child* de n y corresponde a la *i-ésima* visita al nodo hijo.
compute(at) indica que debe evaluarse la ecuación que define at en la producción p aplicada correspondiente al nodo n.
leave(i) indica que ha finalizado la visita *i-ésima* en el nodo corriente y que se debe visitar al nodo padre.

La evaluación termina cuando se ejecutaron todas las operaciones de la secuencia de visita del nodo raíz del árbol sintáctico. En la mayoría de la veces, con una operación *leave*, en muchos casos implícita.

Estos evaluadores pertenecen a una familia denominada *Evaluadores Multi-visita*, ya que el proceso de evaluación puede requerir múltiples visitas a cada nodo para evaluarlo.

2.1. Gramáticas de Atributos Multiplanes

Para abordar la presentación y definición de las Gramáticas de Atributos Multiplanes es necesario introducir algunos conceptos previamente.

Conjunto de dependencias directas de una producción **(DP(p))**: Dada una producción p de una gramática de atributos , entonces

$$DP(p) = \{(X_i.a, X_j.b) | X_i.a \rightarrow Y_j.b \in R^p\}$$

Dependencias entre los atributos de un símbolo **(Down(X))**: Dada un símbolo X de una gramática de atributos definida como en , entonces

$$Down(X) = \{(a,b) | a \rightarrow b\} con\ a, b \in A(X)$$

El siguiente teorema fue presentado por Wuu Yang en [1]:

$$\bigcup_{todo\ p} DCG_X(p) = Down(X)$$

Nota: $\mathbf{DCG_X(p)}$ contiene las dependencias, entre las instancias de la gramática, para el símbolo `X`, acotando el análisis para la producción p y los posibles contextos inferiores.

El **conjunto de dependencias aumentadas** se denota como $ADP(q|p_1, p_2, \ldots, p_k)$ y se define: Sea q una producción de la forma $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$. Sea p_i una producción cuya parte izquierda es X_i $(1 \leqslant i \leqslant k)$.

$$ADP(q|p_1, p_2, \ldots, p_k) = DP(q) \bigcup_{k}^{i=1} DGC_{X_i}(p_i)$$

El conjunto de todas las posibles dependencias aumentadas para una producción q se define como: $SADP(q) = \bigcup_{q \in P} ADP(q|p_1, p_2, \ldots, p_k)$

Definición 1. *Una gramática de atributos G es una gramática de atributos multiplanes si y solo si*

$$\forall q : q \in P : (\forall g : g \; es \; un \; grafo \; de \; q \wedge g \in SADP(q) : g \; es \; no \; circular)$$

3. magGen

El desarrollo de **magGen** se puede dividir en 4 etapas:

- Definición del Lenguaje especificación de MAG.
- Parser del lenguaje, representación interna y chequeos.
- Construcción de grafos y aplicación de algoritmos de cómputo de planes y secuencias de visita.
- Generación de código del evaluador.

Las cuales son reflejadas en el funcionamiento **magGen** ya que se realiza atravesando cada una de estas etapas secuencialmente, es decir, la terminación exitosa de una, habilita la siguiente; por lo tanto cada etapa mantiene su salida de errores de manera independiente.

La salida exitosa de **magGen** que indica que se han realizado todas las etapas correctamente, es la siguiente:

```
* Parsing grammar ---------- [  OK  ]
* Generation graphs --------- [  OK  ]
* Build plans -------------- [  OK  ]
* Build visit sequences ----- [  OK  ]
* Generation code ---------- [  OK  ]

Generation complete in: 0.372814 seconds.
```

En caso de funcionamiento anormal de alguna de las etapas, se detectará un **FAIL** en la etapa correspondiente y se indicará la información del mismo.

Un caso alternativo de salida, se da cuando **magGen** detecta planes cíclicos, visualizándose un **ABORT** en la etapa de creación de planes:

```
* Parsing grammar ---------- [  OK  ]
* Generation graphs -------- [  OK  ]
* Build plans ------------- [ ABORT ]

 ERROR: One o more graph ADP has an cycle in its dependencies. ↩
 Look the folder ./out_maggen/graphs/CYCLIC_graphs/ for more ↩
 details.
```

3.1. Lenguaje de especificación de las MAG

El lenguaje de especificación utilizado para la descripción de una MAG de entrada para **magGen**, fue definido en el marco del proyecto.

El lenguaje se subdivide en secciones bien marcadas que permite la definición de una MAG:

Bloque Dominio Semántico Destinando a la declaración de sort, operadores y funciones que se utilizarán en los otros dos bloques. Este bloque es denominado "`semantic domain`".

Bloque de Atributos Destinando a la declaración y definición de los atributos asociados a cada símbolo. Este bloque es denominado "`attributes`".

Bloque de Reglas Destinado a la declaración y definición de las reglas sintácticas de la gramática con sus correspondientes ecuaciones semánticas para cada atributo asociado a cada símbolo. Este bloque es denominado "`rules`".

Los tres bloques analizados anteriormente, pueden ser clasificados según su comportamiento o funcionalidad dentro del lenguaje la especificación.

Los dos primeros, son bloques puramente declarativos o dedicados a la definición de elementos que serán utilizados en el tercer bloque. La sección de reglas es considerado el de mayor importancia, ya que marca la sintaxis y semántica de la gramática.

Cada bloque del lenguaje de especificación contiene su sintaxis propia para su definición[3].

En la sección 3.2 se presenta un ejemplo de una gramática de atributos multiplanes presentada por Wuu Yang en [1] codificada en el lenguaje de especificación de **magGen**.

3.2. Ejemplo de MAG especificada en el lenguaje de especificación de magGen.

En la figura 1 se presenta un ejemplo de una MAG y en la figura 2 se presenta la traducción de la gramática en el lenguaje de especificación de **magGen**.

3.3. Parser

El parser del lenguaje se apoya en la utilización de un framework reconocido mundialmente, denominado ***Spirit***, perteneciente a la biblioteca de C++ llamada ***Boost***. Esta decisión trajo dos grandes beneficios; la confiabilidad del parser

[3] En la documentación completa de **magGen** se pueden encontrar con detalles el lenguaje de especificación.

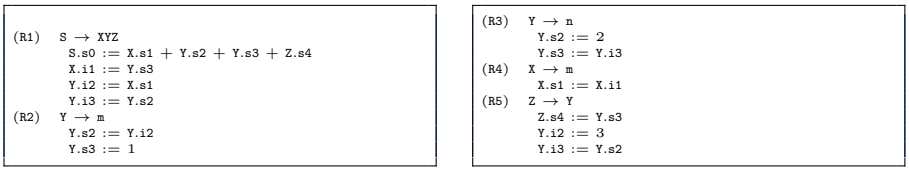

Figura 1. Ejemplo MAG

```
1  semantic domain
2    op infix (10, left) +: int, int -> int;
3  attributes
4    s0: syn <int> of {S};
5    s1: syn <int> of {X};
6    s2: syn <int> of {Y};
7    s3: syn <int> of {Y};
8    s4: syn <int> of {Z};
9    i1: inh <int> of {X};
10   i2: inh <int> of {Y};
11   i3: inh <int> of {Y};
12
13 rules
14   S ::= X Y Z
15       compute
16         S[0].s0 = X[0].s1 + Y[0].s2 + Y[0].s3 + Z[0].s4;
17         X[0].i1 = Y[0].s3;
18         Y[0].i2 = X[0].s1;
19         Y[0].i3 = Y[0].s2;
20       end;
```

```
21   Y ::= 'm'
22       compute
23         Y[0].s2 = Y[0].i2;
24         Y[0].s3 = 1;
25       end;
26   Y ::= 'n'
27       compute
28         Y[0].s2 = 2;
29         Y[0].s3 = Y[0].i3;
30       end;
31   X ::= 'm'
32       compute
33         X[0].s1 = X[0].i1;
34       end;
35   Z ::= Y
36       compute
37         Z[0].s4 = Y[0].s3;
38         Y[0].i2 = 3;
39         Y[0].i3 = Y[0].s2;
40       end;
```

Figura 2. MAG en lenguaje de especificación de **magGen**.

obtenido y la rápida obtención del mismo, ya que, la gran ventaja de ***Spirit*** es permitir escribir la definición de la gramática en lenguaje **C++**.

Spirit es un framework generador de analizadores sintácticos, o parsers, descendentes recursivos orientado a objetos implementado usando técnicas de meta-programación con plantillas. Las expresiones mediante plantillas, permiten aproximar la sintaxis de una "*Forma Backus Normal Extendida*" (**EBNF**) completamente en C++.

3.4. Ciclicidad

El chequeo de ciclicidad sobre los grafos ADP, se implementó utilizando el algoritmo de búsqueda en profundidad (Depth-first search) de ***Boost*** combinado con la creación de un "visitador" especializado. El cual a medida que recorre el grafo, va guardado los nodos y aristas visitadas mientras no se halla detectado un ciclo. Ese subgrafo es mostrado al usuario en caso de error, sino es descartado.

3.5. Construcción de planes

La etapa de construcción de planes en **magGen** se basa en dos puntos:

1. El punto de entrada para el cálculo de los planes esta dado sobre los grafos **ADP**.
2. El cálculo de planes se basa en un orden topológico de la evaluación de los atributos de los símbolos, teniendo en cuenta los distintos contextos posibles.

Un **plan** se compone de una lista de números de ecuaciones a computar. Todo plan determina un orden de evaluación, realizándose la evaluación de izquierda a derecha.

Para el ejemplo presentado en la sección 3.2 se computan 7 planes (uno por cada ADP). Cada plan guarda, además del orden de evaluación de las ecuaciones, el contexto de la producción. Este, permite, en el proceso de evaluación, poder asignar dicho plan a un nodo del AST a decorar.

- {6,2,9,3,5,4,10,1} con $(\lambda|1|4,2,5)$
- {7,4,8,2,9,3,10,1} con $(\lambda|1|4,3,5)$
- {2,9} con $(1|4|\lambda)$
- {6,3,5,4} con $(1|2|\lambda)$
- {7,4,8,3} con $(1|3|\lambda)$
- {6,11,5,12,10} con $(1|5|2)$
- {7,12,8,11,10} con $(1|5|3)$

Donde la notación $(\alpha|P|\beta)$ especifica: el contexto superior (α), la regla corriente (P) y el contexto inferior (β).

3.6. Construcción Secuencias de Visita

El funcionamiento para la construcción de las secuencias, esta dado por la aplicación de un *recorrido sobre los contextos de las reglas*, para la evaluación de cada atributo de los símbolos. Mediante los cuales se van obteniendo las operaciones que componen a la secuencia. Estas acciones tienen la siguiente interpretación:

Compute Valor menor que cero que representa el número de la ecuación a resolver[4].
Visit valor mayor que cero que representa el número nodo hijo a visitar[5].
Leave valor "0".

A modo de ejemplo, se muestra una secuencia de visita generada por **magGen** para el ejemplo analizado en la sección 3.2:

{-7,0,-8} Secuencia de visita computada a partir del plan {7,4,8,3} con $(1|3|\lambda)$. En este caso, se traduce a lo siguiente: **Compute** computar la ecuación 7, **Leave** retornar el control al ambiente de invocación[6] y **Compute** computar la ecuación 8.

Heurística del algoritmo Consiste en evaluar el plan buscando las dependencias de cada ecuación en el contexto del mismo. A continuación se muestran los pasos básicos a tener en cuenta con un ejemplo:

Si se toma uno de los planes calculados por **magGen** para el ejemplo de la sección 3.2:

Tomando el plan {6,2,9,3,5,4,10,1} con $(\lambda|1|4,2,5)$, el cual es purgado a sólo las ecuaciones de la regla corriente (1):{ 2, 3, 4, 1 }.

[4] La ecuación a computar pertenece a la regla del plan corriente.
[5] El nodo hijo esta dado por el contexto de la regla del plan corriente.
[6] El leave retorna el control a la secuencia de visita de contexto superior, es decir, desde donde se invocó a esta secuencia.

1. Se recorre el plan tomando en cada paso una de las ecuaciones. Para este caso particular se comienza con la ecuación 2.
2. Dada la ecuación `i`[7], en el plan, se debe computar todo el *l_value* de la ecuación. Para ello se deben resolver todas las dependencias dadas por *r_value*.
3. Dada una dependencia de la ecuación, primero se chequea si la misma fue computada en algún paso anterior, sino, se analizan los siguientes casos:
   - Si la dependencia proviene de una instancia que pertenece al **símbolo de la parte izquierda de la regla** y además contiene un **atributo heredado**, entonces se debe realizar un "`leave`".
   - Si la dependencia proviene de una instancia que pertenece a un **símbolo de la parte derecha de la regla** y además contiene un **atributo sintetizado**, entonces se debe de realizar un "`visit`". En este caso, se debe analizar, según el contexto, cuál es el plan se debe visitar, es decir, a cuál nodo hijo.

   El caso de cómputo de atributos sintetizados de símbolos de la parte izquierda y atributos heredados de símbolos de la parte derecha, deben resolverse en el ambiente actual, lo cual se puede garantizar por la consistencia del plan de evaluación.
4. Luego de la obtención de todas las dependencias, se realiza un **compute** del *l_value* y se marca a este como `evaluado`. El paso siguiente es avanzar una ecuación en la lista que impone el plan y realizar los mismos tratamientos.

Para el cómputo de las secuencias de visitas, solamente alcanza con lanzar el algoritmo para los planes iniciales[8]. Esto se sustenta con las propiedades de la gramática, principalmente la alcanzabilidad. De esta forma, los planes iniciales lanzan los demás planes necesarios para el cómputo total de las secuencias.

3.7. Generación de código

La etapa final de **magGen** esta dada por la generación de código para el evaluador estático. Esta, produce dos archivos: *interface* (.hpp) e *implementación* (.cpp). Los archivos generados se vinculan con dos módulos estáticos, `Node.hpp` y `Node.cpp`.

La generación de código es un proceso reiterativo sobre cada elemento almacenado en los repositorios de la herramienta. En los archivos antes mencionados se reflejan las reglas de la gramática, los planes de evaluación computados, las secuencias de visita y un conjunto de métodos para la manipular esta información. Dos de estos métodos son los encargados de la evaluación de un AST.

3.8. Algoritmo de evaluación

El algoritmo que se implementó para poder evaluar un AST se basa en dos etapas bien definidas:

[7] Para este caso particular i toma los valores 1,2,3,4.
[8] Planes asociados a la regla inicial.

Primera recorrida del AST se invoca a la función

- `traverse`: responsable de seleccionar el plan de evaluación para cada nodo no terminal. Esta función a partir del identificador de regla que posee cada nodo AST, construye su contexto de invocación y teniendo en cuenta el orden superior impuesto selecciona el plan ha asignar a dicho nodo. Luego de la asignación de un plan al nodo, se prosigue con el tratamiento a los nodos hijos. La función `traverse` teórica, es presentada en la figura 3(A).

Segunda recorrida del AST se invoca a la función

- `eval_visiter`: es un evaluador orientado a visitas, que debe computar los valores de los atributos de los nodos. Esta función sigue los lineamientos de la secuencia de visita referenciada por cada nodo. Arrancando desde la raíz, el algoritmo dirige la evaluación del AST para lograr el AST decorado. La misma es presentada en la figura 3(B).
  Notar que cuando se debe realizar un `visit`, sólo consiste en llamar a el evaluador sobre el hijo indicado. Si en cambio, se debe hacer un `compute`, se llama a la función `compute_eq`, que está encargada de procesar la ecuación. Y cada vez que sucede un `leave`, se debe guardar en el nodo actual la próxima posición de la última visitada, para que en la siguiente visita, el evaluador continúe desde ese punto, omitiendo al `leave`[9].

El código del evaluador es reducido por su gran nivel de modularización. Recibe como parámetro la raíz del AST a evaluar. El orden inicial de evaluación para los atributos del símbolo inicial y, que por ser una gramática extendida no ocurre en ningún otro lado de la misma, siempre se encuentra en la posición cero del arreglo de planes de evaluación únicos.

```
1 procedure eval (T)
2   /* T es un AST no evaluado. */
3   /* μ es un orden de evaluacin de los atributos del simbolo inicial. */
4   /* Selecciona un plan de evaluacin para cada nodo no terminal en T. */
5   traverse(root of T, μ)
6   /* Evaluador Orientado a Visita tradicional para evaluar las instancias de atributos de T.*/
7   eval_visiter
8   /* T se encuentra decorado*/
9 end procedure
```

4. Medidas de performance

En la etapa de prueba de **magGen** se han analizado una serie de medidas performance que permitieron observar números concretos sobre el funcionamiento de la herramienta. En la figura 4 se muestran los tiempos en la compilación de la herramienta, teniendo en cuenta dos versiones de Spirit Boost[10]. Se detecta un incremento de performance con el uso de la Spirit Boost actual. En las figuras 5 se presentan medidas sobre los tiempos y tamaños logrados en el binario obtenido del evaluador generado por **magGen**. Estas medidas son aplicadas a

[9] Con el guardado del estado de cada secuencia de visita para cada nodo, se pudo omitir el índice i, que se presentó en las operaciones de la teoría de Kastens.

[10] Se uso la mínima versión compatible (Boost 1.37) y la actual versión (Boost 1.43).

```
1  procedure traverse (n, ω)
2    /* n es un nodo no terminal node de un AST. ↩
         */
3    /* ω es un orden de evaluacin de los atributos ↩
         del simbolo no terminal de mas a ↩
         izquierda localizado en n.*/
4    q := la produccion aplicada al nodo n
5    Sean p1, p2, . . . , pk las producciones aplicadas a ↩
         los nodos hijos no terminales de n.
6    plan [n] := Γ[q, ω, p1, p2, . . . , pk]
7    for cada hijo no terminal mi of n in T do
8      traverse (mi , Θ[q, ω, i, p1, p2, . . . , pk])
9    end for
10 end procedure
```

(A) Algoritmo Traverse.

```
1  procedure eval_visiter(n)
2    /* n es un nodo no terminal de un AST. */
3    /* Donde n contiene: */
4    /* .v_seq es la secuencia de visita para el nodo ↩
         n. */
5    /* .index es el ultimo elemento tratado de la ↩
         secuencia. */
6    for cada operacion op ∈ v_seq mayor a index do
7      if (op == visit m) then
8        /* m es el hijo a visitar. */
9        eval_visiter(m);
10     else if (op == compute eq) then
11       /* eq es la ecuacion a computar. */
12       compute(eq);
13     else
14       /* Se produce un leave. */
15       /* Se actualiza el ndice del nodo actual. */
16       index = get_index(op);
17       break;
18     end if
19   end for
20 end procedure
```

(B) Algoritmo Eval_visiter.

Figura 3. Algoritmos utilizados en la evaluación.

dos de los ejemplos corrientes en el desarrollo de la herramienta como lo son, `Wuu Yang` (presentado en la sección 3.2) y `Expresiones aritmética` presentado en la figura 6.

Input / Spirit	**1.8.X**	**2.3**
`MAG Wuu Yang`	∼0.087 sec	∼0.084 sec
`MAG Aritmética`	∼0.590 sec	∼0.450 sec

Figura 4. Medidas de performance: Versión de Spirit Boost

Input / g++	**sin -O3**	**con -O3**
`Wuu Yang`	∼1.51 sec	∼2.55 sec
`Aritmética`	∼5.98 sec	∼**1m 5.6 sec**

(a) Tiempos de compilación

Input / g++	**sin -O3**	**con -O3**
`Wuu Yang`	104 Kb	48 Kb
`Aritmética`	516 Kb	384 Kb

(b) Tamaños del ejecutable

Figura 5. Medidas para el evaluador generado

4.1. Unicidad de planes

La unicidad de planes se trata de no asociar cada plan con su dependencias, sino usar una indexación a planes únicos. Esto permite que distintos contextos se asocien al mismo plan, lo que permite una notable optimización tanto en el procesamiento como en la generación de código. En la figura 7 se presenta una comparación de **magGen** utilizando unicidad de planes y sin el uso de la misma para el ejemplo de `aritmética`. Notar que la eficiencia es notable tanto en la generación de código como en la cantidad de secuencias de visita.

5. Conclusión

Se logró obtener una herramienta modularizada, eficiente y completamente desarrollada en C++. Además, no se conocen herramientas que trabajen sobre MAG, en este sentido, **magGen** adquiere mayor utilidad.

Se pueden resumir las características de **magGen** en los siguientes puntos:

```
M ::= E
    M.valor = E.valor
E ::= E '+' E
    E.valor = E.valor + E.valor
    | E '-' E
        E.valor = E.valor - E.valor
    | E '*' E
        E.valor = E.valor * E.valor
    | E '/' E
        E.valor = E.valor / E.valor
    | '(' E ')'
        E.valor = E.valor
    | '-'E
        E.valor = E.valor * (-1.0)
    | num
        E.valor = num.valor
num ::= digit num
          num.valor = (digit.valor * 10.0) + num.valor
    | real
          num.valor = real.valor
      | digit
          num.valor = digit.valor
```

```
digit ::= '1'
          digit.valor = 1.0
        | '2'
          digit.valor = 2.0
        | '3'
          digit.valor = 3.0
        | '4'
          digit.valor = 4.0
        | '5'
          digit.valor = 5.0
        | '6'
          digit.valor = 6.0
        | '7'
          digit.valor = 7.0
        | '8'
          digit.valor = 8.0
        | '9'
          digit.valor = 9.0
        | '0'
          digit.valor = 0.0
real ::= digit '.' digit
          real.valor = digit.valor + (digit.valor / 10.0)
```

Figura 6. GA de Expresiones.

Variable/Unicidad	Sin	Con
Cant. de planes	371	371
Cant. de planes proy.	687	687
Cant. de sec. de visita	371	22
Generación del evaluador	∼1.44 sec	∼0.47 sec
Cant. de líneas (eval.)	17464	5974

Figura 7. Unicidad de planes. Ejemplo expresiones aritméticas

- El lenguaje de especificación para AG tiene una sintaxis transparente, concisa y amplia, facilitando la traducción de cualquier MAG al mismo.
- La herramienta fue desarrollada íntegramente en C++, como así también el evaluador generado.
- El evaluador generado es estático, lo que asegura que no existe un "**overhead**" en el cómputo de la evaluación, ya que el mismo se llevó a cabo mientras se generaba el evaluador propiamente.

Referencias

1. Wuu Yang. 1998. *Multi-Plan Attribute Grammars*. Department of Computer Information Science. National Chiao-Tung University, Hsin-Chu, Taiwan. 2, 3, 5
2. Wuu Yang. 1999. *A Classification of Non Circular Attribute Grammars based on Lookahead behavior*. Department of Computer and Information Science. National Chiao-Tung University, Hsin-Chu, Taiwan.
3. Arroyo, Marcelo Daniel. *Gramáticas de atributos, clasificación y aportes en técnicas de evaluación*. Tesis de carrera de Magíster en Ciencias de la Computación. Universidad Nacional del Sur. Bahía Blanca - Argentina.
4. U. Kastens. 1980. *Ordered Attribute Grammars*. Acta Informática. Vol. 13, pp. 229-256. 2
5. Jazayeri, Ogden and Rounds. 1975. *The intrinsically exponential complexity of the circularity problem for attribute grammars*. Comm. ACM 18. December 2, Pag: 697-706. 2
6. Alfred V. Aho, John E. Hopcroft, Jefrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley publishing Company, Reading, Massachusetts, E. U. A.(1983).

7. D. Knuth. 1968. *Semantics of context free languages*. Math Systems Theory 2.June 2.Pag: 127-145. 1