GAMA-NW: Un enfoque reutilizable para un metalenguaje y un metacompilador

Otto Fernando Pflücker López
Pontificia Universidad Católica del Perú
Facultad de Ciencias e Ingeniería, Especialidad de Informática
Lima, Perú, Lima 32
opflucker@pucp.edu.pe

and

João José Neto
Universidad de São Paulo - Escuela Politécnica
Departamento de Ingeniería de la Computación y Sistemas Digitales,
São Paulo, Brasil, CEP 05508-900
joao.jose@poli.usp.br

Abstract

GAMA-NW is a grammar for the formal and complete description of a programming language in its lexic, syntactic and semantic aspects. The main difference between our approach and existing ones is the reutilization potential of the generated products at the description and implementation levels.
At the description level it was used only one notation for describe independently each aspect mentioned before. This uniformity relies on the *parallel texts* concept, introduced in this article, and the attributed grammars. Each aspect establish one level in the complete description and, because the uniformity in the notation, each level is treated as a sub grammar. Each sub grammar can use symbols of others sub grammars, establishing a reutilization mechanism.
Each sub grammar is implemented applying the structured stack automata theory, obtaining a component named g-filter. The g-filters are implemented independently and connect one to others with great flexibility, establishing a second reutilization mechanism.

Keywords: Metacompilers, grammars, parallel texts, multi-level grammars, programming languages, compilers.

Resumen

GAMA-NW es una gramática para la especificación formal y completa de un lenguaje de programación en sus aspectos léxico, sintáctico y semántico. La principal diferencia de nuestra propuesta frente a las ya existentes está en el potencial de reutilización de los productos generados a nivel de la especificación y de la implementación.
A nivel de la especificación se ha utilizado una notación única para describir independientemente cada aspecto mencionado. Esta uniformidad se apoya en el concepto de *textos paralelos*, presentado en este artículo, y en las gramáticas de atributos. Cada aspecto establece un nivel en la especificación total y, dada la uniformidad en la notación, se trata cada nivel como una subgramática. Cada subgramática puede usar símbolos definidos por otras subgramáticas, estableciéndose así un mecanismo de reutilización.
Cada subgramática es implementada aplicando la teoría de autómatas de pila estructurados, obteniéndose un componente denominado g-filtro. Los g-filtros se implementan independientemente y se conectan con gran flexibilidad unos con otros, estableciéndose así un segundo mecanismo de reutilización.

Palabras clave: Metacompiladores, gramáticas, textos paralelos, gramáticas en múltiples niveles, lenguajes de programación, compiladores.

1. Introducción

Al presente no existe consenso sobre un mecanismo de especificación, formal y completa, de un lenguaje de programación (LP). Una especificación como tal requiere un soporte tanto teórico, llevado a cabo mediante el estudio y formalización de diferentes tipos de gramáticas (como las indicadas en la jerarquía de Chomsky y sus muchas derivaciones), como uno práctico, llevado a cabo mediante la estandarización de notaciones que permitan representar las características expuestas por dichas gramáticas, esto es, una estandarización de los metalenguajes.

Adicionalmente es ya un consenso que la especificación de un LP se divida en 3 aspectos: léxico, sintáctico y semántico [23]. Los aspectos mejor comprendidos a la fecha, y para los que sí existen tipos de gramáticas de amplio uso, corresponden al léxico y al sintáctico. El aspecto léxico se suele manejar mediante gramáticas regulares. Los mejores avances respecto al aspecto sintáctico se han alcanzado con gramáticas libres de contexto y, en menor medida, con gramáticas sensibles al contexto.

Por otro lado no es aún claramente identificable un metalenguaje de amplio uso. En estos es común la especificación por separado (bajo diferentes gramáticas y con diferentes notaciones) de cada aspecto de un lenguaje (los ampliamente conocidos LEX y YACC son un claro ejemplo). Sin embargo se encuentra cierto consenso respecto a la representación del aspecto léxico (con el uso de expresiones regulares) y del sintáctico (con el uso de notaciones como BNF, EBNF, Wirth, EWirth, etc.), no así para el semántico.

La especificación del aspecto semántico suele realizarse en lenguajes de programación, por separado de la formalidad de los metalenguajes, y por tanto sin la rigurosidad que una formalización puede ofrecer como lo demuestra claramente, para los otros dos aspectos, la proliferación de programas de generación automática de reconocedores sintácticos. El contar con formalismos igualmente rigurosos para la semántica permitiría desarrollar programadas de generación automática de un compilador completo, o metacompiladores, que gocen de las mimas ventajas de los reconocedores.

Un mecanismo utilizado para incluir el aspecto semántico en la especificación formal de un lenguaje ha sido la extensión de las notaciones sintácticas ampliamente utilizadas. En mayor o menor medida, estas extensiones permiten distinguir cuáles elementos en la especificación corresponden a la sintaxis y cuáles a la semántica. Lamentablemente estas extensiones suelen abarcar solo la llamada *semántica estática* [20] y son realizadas de manera tal que difieren completamente de la notación original lo que inhibe su reutilización.

En la especificación semántica las *gramáticas de atributos* son visiblemente las más utilizadas. Desde su aparición [13] despertaron gran interés y han sido ampliamente estudiadas e implementadas. Si bien su ámbito más conocido corresponde a la semántica estática, su simplicidad y capacidad expresiva las vuelven potencialmente utilizables en la semántica dinámica.

El presente trabajo expone el desarrollo de un metalenguaje, GAMA-NW, que permita una especificación conjunta de los diferentes aspectos de un LP mediante la conjunción de formalismos ampliamente utilizados, lo que se espera facilite su comprensión y uso. Además, esta especificación debe abarcar la semántica completa del lenguaje, utilizar un único formalismo para todos los aspectos del mismo y facilitar la reutilización de los productos obtenidos.

2. Formalización de la sintaxis y de la semántica

La especificación completa de un LP requiere que se especifique cada fase del procesamiento de un programa escrito en dicho LP: Análisis léxico, análisis sintáctico y análisis semántico. Mientras que existe un consenso para la especificación formal de las dos primeras fases [19], el tema es más controversial respecto a la fase semántica.

La formalización del léxico y de la sintaxis son problemas más simples que la formalización de la semántica [20]. El léxico determina cuales símbolos son válidos para un LP y la sintaxis determina las secuencias válidas de dichos símbolos. Por otro lado la semántica determina el significado (lo que puede variar según la interpretación que se le dé a este término) de un programa en un LP. El léxico y la sintaxis tiene que ver con la forma de un programa mientras que la semántica con el significado y, como es fácil intuir (y en la práctica ocurre así), es mucho más simple ponerse de acuerdo en lo primero que en lo segundo. Como consecuencia de esto los progresos en las primeras han venido antes y más rápido que en la última. Muchos lenguajes (como Algol) fueron definidos a través de una descripción léxica y sintáctica formal y una descripción semántica informal. Por ejemplo, es común leer especificaciones de lenguajes como Pascal, Modula, Oberon, Lisp, etc., donde la semántica está descrita como una implementación en el mismo lenguaje especificado, lo que

constituye una especificación informal aunque muy útil.

Formalización implica principalmente *no-ambigüedad*, lo que tiene consecuencias importantes en la implementación de LP's. Por ejemplo, el problema general de decidir si una gramática es o no ambigua es teóricamente irresoluble, pero en la práctica la ambigüedad puede ser evitada, especialmente si nos restringimos a ciertas subclases de gramáticas, como ocurre con algunas subclases de gramáticas regulares (GR) y de gramáticas libres del contexto (GLC). Como consecuencia de ello, las GLC's son actualmente las herramientas estándar usadas en manuales de referencia para definir la sintaxis de lenguajes de programación [19]. Las definiciones léxicas que los acompañan son usualmente dadas usando *expresiones regulares* (ER). El significado práctico de estos formalismos es aumentado enormemente por el hecho de que ellos permiten implementar automáticamente procesadores de lenguajes eficientes, analizadores sintácticos a partir de GLC's y analizadores léxicos a partir de ER's. El fácil intuir que llegar a un consenso en la formalización de la semántica tendrá consecuencias similares.

A falta de un consenso para la formalización de la semántica, una variedad de formalismos diferentes ha sido propuesta permitiendo en algunos casos la generación automática de implementaciones completas de LP's desde dichas definiciones.

Es importante en este punto establecer la separación que en el presente trabajo se adopta para los aspectos que forman parte de la sintaxis de un LP y los que forman parte de su semántica (debido también a una falta de consenso entre los teóricos). Se ha adoptado la posición establecida por Pagan [20]:

- **sintaxis** Todos los aspectos de un LP que pueden *razonablemente* ser explicados solamente en términos de secuencias de símbolos.
- **Semántica** Todos los demás aspectos de interés.

Por tanto, todos los aspectos lingüísticos, fuera de la generación de código y la optimización, que son normalmente manejados por el procesador en tiempo de compilación son *sintácticos en naturaleza* y todos los aspectos normalmente manejados en tiempo de ejecución son *semánticos en naturaleza*. Por ejemplo, la *verificación estática de tipos* se considera parte del aspecto sintáctico del lenguaje y la *verificación dinámica de tipos* parte del aspecto semántico. A este respecto existen dos términos comúnmente utilizados en la literatura:

- **Semántica estática** Aspectos manejados en tiempo de compilación y que no son libres del contexto.
- **Semántica dinámica** Aspectos manejados en tiempo de ejecución.

Por tanto para el presente trabajo se considera a la *semántica estática* [11] como incluida en nuestra sintaxis, y a la *semántica dinámica* como incluida en nuestra semántica.

Cualquier formalización semántica debe, en algún sentido, asociar *significado* con los programas y construcciones de un *lenguaje sujeto*. Hay una vasta variedad de aproximaciones, las que se suelen clasificar en tres grupos: Operacionales, denotacionales y axiomáticas.

En la *aproximación operacional* de la formalización semántica el significado es descrito con dispositivos tales como máquinas abstractas con estados discretos y secuencias más o menos explícitas de operaciones computacionales. Esta aproximación el probablemente la más usada y en la que se encuentran las gramáticas de atributos [19].

3. Marco Teórico

En esta sección se presentarán brevemente los conceptos de gramática de atributos, autómatas de pila estructurados y textos paralelos, conceptos en los que se sustenta el presente trabajo.

3.1. Gramáticas de Atributos

Para una gramática de atributos, el *significado* del programa corresponde a los valores de los atributos asociados al nodo raíz del árbol de atributos. Las gramáticas de atributos (GA) fueron originalmente definidas [13] para especificar e implementar los aspectos semánticos (estáticos) de los lenguajes de programación. Desde entonces han tenido una intensa investigación. Por el lado teórico se han definido varias subclases de GA con algoritmos de implementación avanzados. Por el lado práctico se han creado un gran número

de sistemas automatizados basados en GA [1, 2]. Se puede afirmar por tanto que las GA constituyen un formalismo maduro.

Las GA son una extensión natural de las GLC por lo que su implementación puede ser muy eficiente. Por otro lado son menos formales y expresivas que otros métodos más matemáticos (denotacionales y axiomáticos). La evolución de las ideas detrás de este formalismo es resumida en [14].

Existen una amplia clasificación de las gramáticas de atributos [4, 10, 2]: canónicas, S-atribuidas (GA-S), L-atribuidas (GA-L), absolutamente no-circulares, circulares, ordenadas, remotas, de referencia, de alto-orden, etc. También se han desarrollado nuevas GA's combinando dos o más GA's previamente definidas como las *de referencia y circulares* [15], o con otras técnicas como las *de referencia rescribibles* [6].

Algunas subclases, aunque aparentemente muy primitivas, se han vuelto sorprendentemente prácticas en un gran número de generadores de procesadores de lenguajes. Por ejemplo, YACC, probablemente la herramienta de implementación de lenguajes más conocida, puede ser vista como la implementación de una gramática de atributos *S-atribuida*. Otra motivación en el estudio de estas subclases es que pueden ser vistas como modelos de procesamiento de lenguajes de uno o más pasos. Por ejemplo, las GA-S y GA-L pueden verse como modelos de compilación de un solo paso, mientras que las GA ordenadas y absolutamente no-circulares pueden verse como modelos de compilación de múltiples pasos.

3.2. Autómatas de Pila Estructurados

Los autómatas de pila estructurados (APE), presentados por primera vez en [18], son una modificación del formalismo original para autómatas de pila. Esta modificación consiste en incluir transiciones entre estados con símbolos no-terminal de la gramática reconocida por el autómata y no solo con símbolos terminales. Una transición con un símbolo no-terminal es resuelta mediante una llamada a una submáquina específica que reconoce dicho no-terminal. Para una gramática dada, se tiene un autómata de pila estructurado que reconoce todo el lenguaje definido por la gramática. Tal autómata se constituye de submáquinas, cada cual especializada en el reconocimiento de un particular no-terminal de la gramática. Así, al no-terminal inicial de la gramática le corresponde la submáquina principal del autómata, es decir, aquella que deberá ser llamada en primer lugar. Las demás submáquinas serán activadas siempre que cualquier submáquina del autómata (incluso ella misma) ejecute alguna transición de estados con un no-terminal. En tal situación, y solamente en ella, la pila del autómata es utilizada: antes de la activación de la submáquina llamada, el autómata apila dos informaciones: la submáquina donde se ha originado la llamada, y el estado de retorno. Entonces, el control pasa para la submáquina llamada, desde su estado inicial. Si tal submáquina alcanza alguno de sus estados finales y no tiene más transiciones posibles para ejecutar, se dice que la submáquina ha logrado reconocer el no-terminal que representa. Cuando esto ocurre la submáquina puede terminar su operación y retornar a la submáquina que la activó (o submáquina llamadora) devolviéndole así el control. La submáquina llamadora continuará desde el estado de retorno previamente memorizado en la pila. Naturalmente, si dado un símbolo de la cadena de entrada, una submáquina actual y un estado actual, no hubieran transiciones posibles y la submáquina actual no estuviera en uno de sus estados finales, un error sintáctico será detectado y el autómata detendrá su ejecución y rechazará su cadena de entrada.

Es posible probar que cualquier autómata de pila, denotado de la manera convencional, puede ser representado a través de esta notación alternativa y viceversa, lo que demuestra la equipontencia de las dos notaciones [18].

Desde la perspectiva del presente proyecto los APE ofrecen una posibilidad interesante de reusabilidad que es explotada por GAMA-NW: Dado que cada no-terminal de una gramática es reconocido por una máquina y la especificación completa de esta es independiente de las submáquinas a la que requiere llamar entonces los símbolos no-terminales correspondientes a las transiciones de dicha máquina puede estar definidos tanto en la misma gramática como *en cualquier otra*. En otros términos, este formalismo facilita significativamente la reutilización de los símbolos de una gramática en otra, característica que es la base del sistema de gramáticas multi-nivel planteado en el presente proyecto.

3.3. Textos Paralelos

Definición: Un *texto paralelo de orden n* es un texto donde pueden identificarse n alfabetos $\sum_i', (i = 1, ..., n)$ y se organiza como una secuencia ω de n-tuplas

$$\omega = (\sigma_{1,1}, \sigma_{2,1}, ..., \sigma_{n,1})(\sigma_{1,2}, \sigma_{2,2}, ..., \sigma_{n,2})...(\sigma_{1,p}, \sigma_{2,p}, ..., \sigma_{n,p})$$

Donde $con|\omega| = p; \sigma_{i,k} \in \Sigma'_i \cup \{\epsilon\}; i = 1, ..., n; k = 1, ..., p$. En general, cualquier $\sigma_{i,k}$ puede ser algún elemento de un lenguaje bien definido, es decir, Σ'_i puede definirse como algún lenguaje sobre su correspondiente alfabeto Σ_i.

Un lenguaje Σ'_i cuyas sentencias tienen longitud unitaria sobre su correspondiente alfabeto corresponde al caso estructuralmente más simple. En esta situación, la secuencia $\omega^0 = \sigma_{1,1}\sigma_{1,2}...\sigma_{1,p}$ se denomina *secuencia de base* o *secuencia de referencia* del texto paralelo ω.

Así, eligiéndose un elemento $\sigma_{1,k}(1 \leq k \leq p)$ de la secuencia de base, se determinan sus correspondientes elementos $\sigma_{j,k}$ de orden $1 \leq j \leq n$. Por tanto, se puede considerar que $\sigma_{1,k}$ es su propio correspondiente de orden 1.

En el caso particular de un texto paralelo cuyo $\sum'_i$ es un lenguaje cuyas sentencias tienen longitud unitaria sobre su correspondiente alfabeto, cualquier elemento $(\sigma_{1,j}, \sigma_{2,j}, ..., \sigma_{n,j})(1 \leq j \leq p)$ tendrá sus componentes $\sigma_{i,j}(1 \leq i \leq n, 1 \leq j \leq p)$, todos con el formato básico de un elemento de su correspondiente alfabeto.

Ejemplo: Textos paralelos de correspondencia entre una secuencia de referencia, definida como un texto en portugués, y las transcripciones fonéticas para n variantes de pronunciación, referentes al portugués hablado en locales diversos de Portugal, Brasil, Angola, Mozambique, Macao y otros países donde se habla el portugués. Para ese caso, Σ'_1 es la colección de todas las palabras ortográficamente correctas de la lengua portuguesa, y los $\Sigma'_k(1 \leq k \leq n)$ son las correspondientes transcripciones fonéticas en los n dialectos deseados.

En un contexto más general, se puede formar una secuencia de base con elementos con longitud no unitaria. En tal situación, el análisis no se hace símbolo a símbolo sino, con el auxilio de un preprocesador, que ejecute una función similar a la de los analizadores léxicos en un compilador, el cual se encarga de la extracción de la secuencia de símbolos elementales que componen cada elemento de la secuencia de base.

Similarmente, para cualquier Σ'_i que no sea formado de cadenas unitarias, un correspondiente preprocesador ha que ser usado para la extracción de cadenas pertinentes al lenguaje asociado.

Así, en general, se puede extraer cada una de los n componentes de cada n-tupla que forma la secuencia ω de un texto paralelo mediante la intervención de un proprocesador (o analizador léxico) correspondiente.

Ejemplo: Un texto paralelo de orden $n = 4$, con: Σ'_1 = la notación de Wirth; Σ'_2 = la secuencia de atributos por defecto correspondiente a cada elemento de la notación de Wirth; Σ'_3 = la secuencia de acciones semánticas asociadas a cada uno de esos elementos; Σ'_4 = la secuencia de atributos cuyo nombre ha sido elegido por el usuario. En tal caso, tenemos:

- $\omega = (\sigma_{1,1}, \sigma_{2,1}, \sigma_{3,1}, \sigma_{4,1})...(\sigma_{1,p}, \sigma_{2,p}, \sigma_{3,p}, \sigma_{4,p}) \in (\Sigma'_1)^*$
- $\Sigma'_1 = \{\omega^0 = \sigma_{1,1}\sigma_{1,2}...\sigma_{1,p} \mid p \geq 1; \omega^0$ es un texto correcto en la notación de Wirth $\}$.
- Σ'_2 = para cada símbolo $\sigma_{1,k}$ en ω^0:
  - si $\sigma_{1,k}$ es algún meta-símbolo, entonces $\sigma_{2,k} = \epsilon$.
  - si $\sigma_{1,k}$ es un terminal o un no-terminal, entonces $\sigma_{2,k}$ = un identificador (único) de atributo.
- Σ'_3 = para cada símbolo $\sigma_{1,k}$ en ω^0:
  - $\sigma_{3,k}$ es una secuencia de expresiones que envuelven los identificadores de atributos que han sido elegidos por el usuario.
- Σ'_4 = para cada símbolo $\sigma_{2,k} \neq \epsilon$, $\sigma_{4,k}$ es un identificador único, elegido por el usuario.

4. Propuesta

La propuesta de solución abarca dos ámbitos: La especificación del metalenguaje GAMA-NW y el metacompilador que lo soporta. En cada ámbito se establece un mecanismo de reutilización. En las subsecciones siguientes se detallan las estrategias seguidas y los conceptos utilizados en cada caso para conseguir productos reutilizables.

4.1. El metalenguaje

La estrategia seguida en el diseño del metalenguaje GAMA-NW es la de uniformizar las notaciones utilizadas para la especificación de cada aspecto de un lenguaje de programación. Esta uniformización se apoya en el concepto de *textos paralelos* ya expuesto.

Para ello una especificación completa de una gramática se hace corresponder con un conjunto de *textos paralelos*. Cada *texto* posee una gramática propia, por lo que podemos hablar de *gramáticas paralelas* o *gramáticas multi-nivel*. De esta forma una gramática completa puede ser especificada por partes, dividiendo dicha especificación en un conjunto de subespecificaciones, cada una correspondiente a un subgramática. Dada la división ampliamente aceptada de una gramática en tres aspectos, léxico, sintáctico y semántico, y que dichos aspectos suelen especificarse por separado, involucrando cada uno su propia gramática, podemos hablar de una gramática total cuya especificación es dividida en tres subgramáticas, una subgramática léxica, una sintáctica y una semántica. En consecuencia, si la especificación de cada subgramática se realiza de manera uniforme entonces tendremos un mecanismo de reutilización definido.

Para ayudar a aclarar estas ideas se ha tomado una variante de un lenguaje simple, al que denominaremos CALC, presentado en [24] y [19], y que corresponde a una calculadora para expresiones aritméticas que involucren constantes numéricas con nombre. En CALC se pueden escribir programas de la forma:

"let" DEFINITIONS "in" EXPRESSIONS "end"

Cada constante aplicada en `EXPRESSIONS` debe estar definida en `DEFINITIONS`, y `DEFINITIONS` no debe dar múltiples valores para una constante. Por ejemplo, el siguiente es un programa escrito en CALC:

```
let
    var a = 10
    var b = 5
in
    a + 4 + b + 8
end
```

Para mantener concisa la especificación del lenguaje solo se permiten sumas en `EXPRESSIONS`. Una selección mayor de operadores solo haría la gramática más extensa sin mostrar ninguna característica nueva. Sin embargo, CALC mantiene muchas características de los lenguajes de programación reales, como son:

- Declaración de entidades con nombre (constantes numéricas en nuestro caso).
- Uso de entidades con nombre.
- Condiciones en la declaración y uso de la entidad:
  - Ninguna constante puede declararse múltiples veces.
  - Solo las constantes declaradas pueden ser referidas por nombre.
- Entidades ejecutables (las expresiones en nuestro caso).

También es posible considerar a CALC como parte de la especificación completa de un lenguaje de programación más avanzado. La siguiente es la especificación léxica de CALC utilizando GAMA-NW:

```
calc_lexic = {

  attributes = { text : string, value : integer }

  productions = {
    ID = LETTER             @ ID.text = LETTER.text @
        { LETTER            @ ID.text = ID.text + LETTER.text @
          | DIGIT           @ ID.text = ID.text + DIGIT.text @
        } .                 ? ID.text in ("let", "var", "in", "end") : iskeyword ?

    INTEGER = DIGIT         @ INTEGER.value = DIGIT.text @
        { DIGIT             @ INTEGER.value = INTEGER.value * 10 + DIGIT.text @
        } .

    DIGIT = '[0-9]'.
```

```
    LETTER = '[a-zA-Z]'.

    SEPARATOR = '[ \t\n]'.  ? true : ignore ?
  }

}
```

Esta especificación le asigna el nombre `calc_lexic` a la gramática descrita. En ella existen 3 símbolos no-terminales que no son utilizados dentro de una regla sintáctica, por lo que son los símbolos principales de esta gramática y los que generaría un analizador léxico que implemente esta gramática. El signo `@` delimita una *regla semántica de asignación* mientras que el sígno `?` delimita una *regla semántica de condición*.

Nótese que, a diferencia de las notaciones convencionales para GA's, aquí las reglas son aplicadas al nivel de los símbolos dentro de la regla sintáctica. Esto se corresponde con la descripción dada para los textos paralelos y permite además el utilizar líbremente los *bucles* y las *alternativas* propias de la notación de Wirth. Nótese además que las reglas semánticas de condición no solo sirven para describir condiciones que al no cumplirse involucren la generación de un error, sino además para definir otras propiedades de los no-terminales reconocidos. Por ejemplo, al finalizar la identificación de un `ID` se verifica que su atributo `text` esta dentro (`in`) de un conjunto de valores, en cuyo caso el `ID` es un `keyword` y debe ser emitido por un analizador léxico como un terminal. Otro ejemplo es el no-terminal `SEPARATOR` que tiene una regla condicional cuya expresión booleana siempre se cumple (`true`) y por tanto siempre se ejecuta la instrucción `ignore`. Es así como se indica que dichos símbolos debe reconocerse pero ignorarse. Por último, nótese el uso de expresiones regulares entre comillas simples. Dichas expresiones corresponden a la definición de un *juego de caracteres* y permite simplificar las reglas sintácticas que, de otra forma, tendrían que detallar uno a uno cada caracter que corresponde a una alternativa para la producción correspondiente.

La siguiente es la especificación sintáctica de CALC utilizando GAMA-NW:

```
calc_sintax = {

  inputs = { calc_lexic }

  attributes = { value : integer, var : collection, vars : collection, code : collection }

  productions = {

    PROGRAM =
        "let" DECLARATIONS    @ EXPRESSION.vars = DECLARATIONS.vars @
        "in" EXPRESSION       @ PROGRAM.code = EXPRESSION.code      @
        "end" .

    DECLARATIONS = VAR_DECL   @ DECLARATIONS.vars += VAR_DECL.var @
        { VAR_DECL            @ DECLARATIONS.vars += VAR_DECL.var @
                              ? VAR_DECL.var in DECLARATIONS.vars : error ?
        } .

    VAR_DECL =
        "var" VAR_NAME
        "=" INTEGER           @ VAR_DECL.var = (VAR_NAME.text, INTEGER.value) @
        .

    EXPRESSION =              @ EXP_FACTOR.vars = EXPRESSION.vars                  @
        EXP_FACTOR            @ EXPRESSION.code += ("LOAD", EXP_FACTOR.value)      @
        { "+" EXPRESSION      @ EXPRESSION[0].code = EXPRESSION[0].value + ("ADD") @
        } .

    EXP_FACTOR = VAR_NAME     @ EXP_FACTOR.value = getvalue(VAR_NAME.text, EXP_FACTOR.vars) @
```

```
                              ? not exist(VAR_NAME.text, EXP_FACTOR.vars) : error          ?
        | INTEGER             @ EXP_FACTOR.value = INTEGER.value                           @
        .

    VAR_NAME = ID .           @ VAR_NAME.text = ID.text @
  }
}
```

La especificación anterior incluye una sentencia que indica que los terminales de entrada se obtendrán utilizando la gramática `calc_lexic` y, al igual que esta, se le da un nombre a la gramática descrita, `calc_sintax`. Al igual que las dos subgramáticas anteriores, la subgramática semántica también es susceptible de ser especificada completamente utilizando GAMA-NW.

En este punto es importante indicar algunas limitaciones actuales del metacompilador de GAMA-NW. Si bien el deseable que el metacompilador pueda autocompilarse (bootstrapping), la implementación actual aún no lo permite. Otra limitación es que la actual clase de GA soportada corresponde a la denominada *l-atributída* y existen otras clases de GA's más avanzadas (como las GA remotas) que permiten una mayor flexibilidad de trabajo al usuario de la gramática. Por último, las pruebas realizadas hasta el momento solo corresponden a lenguajes simples cuya complejidad no es mucho mayor que la del lenguaje CALC presentado.

Se espera que las limitaciones indicadas y nuevas características sean agregadas paulatinamente al metacompilador de GAMA-NW.

4.2. El metacompilador

Dado que el presente trabajo busca enfocarse exclusivamente en la especificación del metalenguaje y no en otros aspectos involucrados en la compilación de un programa, como son la generación y la optimización de código, se ha elegido la CLR (Common Language Runtime) de la plataforma .NET como máquina objetivo para las pruebas de compilación debido principalmente a que simplifica significativamente el trabajo de generación de código [8].

La arquitectura propuesta para el metacompilador corresponde al estilo arquitectónico *Tuberías y Filtros* [7]. Un filtro es un componente con un conjunto de entradas y un conjunto de salidas. El filtro lee datos de sus entradas, los procesa y envía los resultados a sus salidas. La salida de un filtro se puede conectar a la entrada de otro filtro a través de una tubería, la que corresponde a cualquier mecanismo de comunicación entre componentes de software. De esta forma, un sistema completo puede ser construido mediante la conexión de un conjunto de filtros. Las características importantes de este estilo, y por las cuales fue elegido, son:

- Los filtros son entidades independientes unos de otros.
- Los filtros desconocen la identidad de los otros filtros a los que se conectan.

Estas características facilitan la implementación y, más importante aún, la reutilización de los filtros. Esta última propiedad es clave para la solución propuesta pues se busca que las especificaciones gramaticales realizadas mediante GAMA-NW sean reutilizables. Ahora bien, podemos especificar un tipo particular de filtro capaz de leer de su tubería de entrada un flujo de símbolos, procesarlos de acuerdo a las reglas de una gramática dada, reconocer una ocurrencia del símbolo no-terminal inicial de la misma y enviar dicho símbolo a su tubería de salida. Si la gramática reconocida por el filtro es una GA entonces el símbolo enviado a la tubería de salida puede contener atributos que representen la semántica generada durante el procesamiento de los símbolos de entrada. En consecuencia tendríamos un artefacto de software (un tipo de filtro) que implementa completamente una gramática y que además es reutilizable. Denominaremos *g-filtro* a este tipo particular de filtro de aquí en adelante para enfatizar la estrecha relación que se establece entre una gramática y un filtro.

Los g-filtros son la base de la arquitectura del metacompilador de GAMA-NW. La figura 1 muestra como estos conceptos se relacionan en una topología típica.

Como se aprecia en la figura 1, cada g-filtro procesa un flujo de símbolos y emite un flujo de símbolos. Los símbolos de salida (no-terminales) de un g-filtro pasan a ser símbolos de entrada (terminales) del siguiente. Es importante tomar en cuenta que, en conjunto, una configuración de g-filtros implementa una única gramática. Podemos relacionar esto con el trabajo típico de un compilador en 3 etapas, como se muestra en la figura 2.

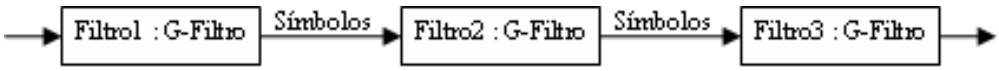

Figura 1: Topología de una arquitectura de *tuberías y filtros* aplicada a g-filtros

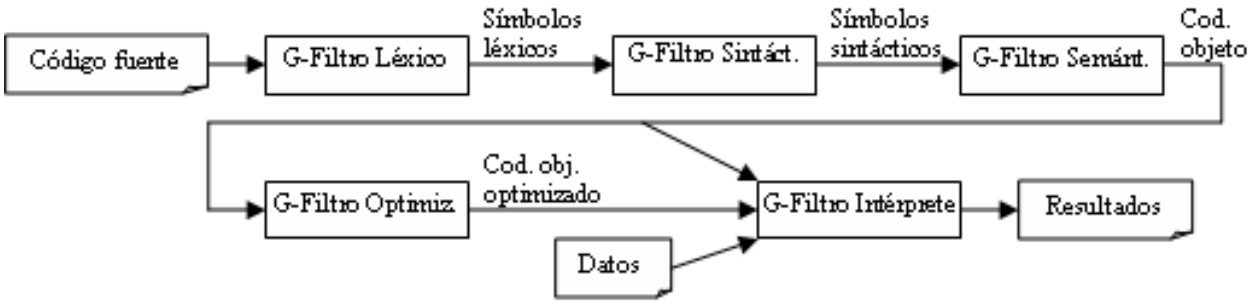

Figura 2: G-filtros aplicados a las 3 fases de un proceso de compilación

Cada fase de la compilación es manejada por un g-filtro, por lo que tenemos un g-filtro léxico, uno sintáctico y uno semántico. Dado que el código fuente se suele leer de un archivo de texto se considera la existencia de un g-filtro especializado que conecte dicho archivo con el g-filtro léxico. Es posible tener además otros g-filtros especializados para otras fuentes de datos, por ejemplo, un puerto de comunicaciones.

En la figura 2 el g-filtro léxico implementa una GR recibiendo como entrada un flujo de símbolos terminales de un solo caracter cada uno y emitiendo como salida un flujo de símbolos no-terminales que corresponden a unidades léxicas como números e identificadores. De forma similar, el g-filtro sintáctico implementa una GA recibiendo como entrada un flujo de símbolos léxicos y emitiendo como salida un flujo de símbolos no terminales y que corresponden al símbolo inicial de la gramática. Estos símbolos pueden incluir atributos con valores adecuados para ser utilizados por el g-filtro semántico, como por ejemplo un árbol de derivación correspondiente al programa reconocido. El filtro semántico puede realizar los análisis necesarios para generar el código objeto y enviarlo, como un atributo de un símbolo, al filtro optimizador. El filtro optimizador tiene como función obtener un código equivalente al recibido, pero optimizado (más eficiente, compacto o adecuado a la máquina objeto donde se ejecutará), y enviarlo al filtro intérprete. La función del filtro intérprete es de ejecutar el código recibido. Este puede ser escogido de entre el código optimizado y el no-optimizado y utilizando los datos de entrada que alimentan la ejecución produciendo un conjunto de resultados. Este último filtro lo representa el CLR [16], la máquina objeto para el presente proyecto.

Es importante señalar en este punto que las dos desventajas principales del estilo arquitectónico *tuberías y filtros* [3, 22] no ocurren en esta implementación. La primera desventaja, el problema de *copiar una gran cantidad de datos de un filtro hacia otro*, no ocurre puesto que lo que realmente se pasa aquí son solo referencias a objetos reservados en *memoria gestionada* por la máquina virtual del CLR y por consiguiente tampoco es un problema la gestión de dicha memoria [21, 16]. Dado que se tiene una *memoria gestionada* única compartida por todos los filtros entonces dicha memoria funciona como un repositorio cuyo uso es la solución típicamente sugerida para el problema de los datos compartidos entre los filtros.

La segunda desventaja de los filtros, el verse obligado a usar siempre el *máximo-común-divisor respecto a la estructura de datos transmitida entre un filtro y otro*, tampoco ocurre en la presente propuesta puesto que cada filtro corresponde necesariamente a una gramática (a excepción del primero y del último) y estas operan de la misma forma, recibiendo símbolos y emitiendo símbolos, por lo que el trabajar con este único tipo de dato no limita el desarrollo de nuevos filtros.

Ahora bien, dado que todos los g-filtros operan de la misma forma, esto es, implementan una gramática, y dado que todas estas implementaciones se realizan utilizando APE's, es por tanto posible simplificar significativamente el trabajo de implementación creando una única biblioteca de software que implemente un tipo de dato g-filtro e instanciándolo varias veces. Cada instancia del tipo g-filtro puede leer la especificación de su gramática de un archivo externo. Por tanto la configuración típica que implemente una gramática en *varios niveles* sería similar a la mostrada en la figura 3.

Se ha utilizado como formato de almacenamiento de la especificación de la gramática a la que accede un tipo g-filtro un esquema XML correspondiente a una adaptación del formato propuesto en [9] por el

Figura 3: Varias instancias de una misma implementación de g-filtro, cada una accediendo a su propia gramática

Vaucanson Group. El uso de un formato XML presenta importantes ventajas de portabilidad y facilidad de extensión del contenido sin afectar los programas preparados para procesar la versión original del formato.

Finalmente, es importante considerar que la arquitectura propuesta permite la posibilidad de que una gramática utilice símbolos provistos por más de una gramática de *nivel inferior*, lo que equivale a decir que es perfectamente viable tener un g-filtro que lea selectivamente símbolos de entrada de más de un g-filtro precedente, como lo muestra la figura 4. Esto de hecho ocurre en las notaciones utilizadas para representar las GA. En ellas se *extiende* una notación originalmente para GLC (como BNF) agregándole *nuevos símbolos*.

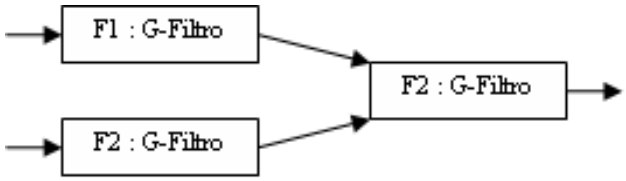

Figura 4: Un g-filtro con más de un g-filtro precedente

Por tanto, la gramática correspondiente a la GLC original puede especificarse por separado de la gramática que corresponde únicamente a la representación de las reglas semánticas y en consecuencia crear un g-filtro para cada una y conectarlos como se muestra en la figura 4. En dicha figura, el g-filtro F3 implementa una GA. La gramática propia de las reglas semánticas es implementada por F1 y el proveedor de símbolos léxicos es F2 (que también podría ser proveedor de F1). Esto muestra la flexibilidad del esquema elegido y la potencialidad de reutilización que provee.

5. Conclusiones

En este artículo se ha presentado la gramática GAMA-NW. En las decisiones de diseño de la solución se han tomado diferentes decisiones de diseño conducentes a lograr el objetivo de generar productos reutilizables.

Por un lado, la gramática GAMA-NW divide la especificación de una gramática compleja en la especificación de varias subgramáticas más simples. Esta división se apoya en el concepto de *textos paralelos*. Por tanto cada subgramática puede ser vista como un *texto paralelo* cuya especificación se apoya en otro texto paralelo (otra subgramática). Además, cada subgramática es especificada de la misma forma, esto es, utilizando la misma notación. Esta división de la especificación permite un primer nivel de reusabilidad, esto es, reusabilidad al nivel de la especificación.

La uniformidad en la especificación de cada subgramática permite aplicar, para la implementación de cada una de ellas, el mismo formalismo, los APE's. Los símbolos procesados y reconocidos conllevan información semántica, la cual es manejada mediante atributos a la manera de las GA's. Además, debido a que se aplica el mismo formalismo en la implementación de cada subgramática, es posible abstraer cada una de estas en un g-filtro y conectar dichos g-filtros de manera que correspondan a las relaciones de dependencia entre las subgramáticas, obteniéndose un compilador completo. Dada la flexibilidad propia de los g-filtros (debido a que su implementación se apega al estilo arquitectónico denominado *tuberías y filtros*) es posible reutilizar cada g-filtro de manera independiente, y sin necesidad de modificación alguna, en más de un compilador. Esto nos proporciona reusabilidad al nivel de la implementación.

Si bien este trabajo muestra la viabilidad de obtener productos reutilizables del proceso de especificación de la gramática de un lenguaje de programación, el metalenguaje GAMA-NW y su metacompilador aún

distan de ser maduros. Diversas limitaciones mencionadas deberán ser superadas y las especificaciones de lenguajes más vanzados deberán ser realizadas.

Quizá debido a que el desarrollo de gramáticas de lenguajes de programación y compiladores es un área bastante especializada y solo un reducido grupo de personas (dentro de toda la comunidad tanto de investigadores como de ingenieros y técnicos en las diferentes áreas de las ciencias de la computación) dedica esfuerzo en ella, la tendencia actual (claramente apreciada en el desarrollo de software utilizado masivamente) de buscar la reusabilidad [22] no ha calado aún significativamente en los productos obtenidos. Sin embargo, algunos esfuerzos en dicho sentido pueden apreciarse [17, 9]. Se espera que el presente trabajo contribuya en dicho sentido.

6. Trabajos futuros

Además del derrotero impuesto por la superación de las limitaciones ya mencionadas, el uso de APE's abre la posibilidad de extender el presente trabajo a la aplicación de un formalismo que deriva de las APE's: Los autómatas adaptativos (AA's) [12]. Los AA's muestran la característica de cambiar dinámicamente ellos mismos en respuesta a un estímulo de entrada sin interferencia de agentes externos [5]. Dada las características de gramática GAMA-NW, se hace posible, mediante ella, la definición de un lenguaje de programación que incluya un mecanismo de adaptación como parte de su especificación. Hay dos elementos que apoyan esta iniciativa:

- El metacompilador para GAMA-NW puede servir también como compilador del lenguaje sujeto. Tanto el metacompilador como el compilador pueden implementarse como una secuencia de g-filtros conectados.
- La maquina objeto elegida, CLR, permite no solo realizar compilación en-línea, sino además modificar el código ya compilado antes de que este sea pasado, en ejecución, al compilador JIT, generarse el código máquina nativo y ejecutarse.

Un programa escrito en un lenguaje adaptativo como el indicado podría incluir referencias a archivos que especifiquen extensiones de una gramática básica. En ejecución, cuando el compilador alcance dichas referencias ejecutaría, sobre los archivos referidos, el metacompilador GAMA-NW (esto es, instanciaría los g-filtros necesarios para procesar los archivo con las especificaciones gramaticales) y generaría una librería (esto es, un nuevo g-filtro que reconozca los nuevos símbolos), la cual sería enlazada dinámicamente con el g-filtro correspondiente del compilador (esto es, agregar algunas transiciones en los autómatas adecuados). Finalmente, es posible mejorar la eficiencia del compilador final obtenido utilizando la compilación en-línea que provee la máquina objetivo CLR para generar, en tiempo de compilación, las extensiones al compilador actual como librerías nativas de la CLR.

Entre las aplicaciones prácticas de una técnica como la descrita esta la capacidad de extender la sintaxis de un lenguaje. Un ejemplo claro de esto se puede apreciar en los lenguajes comerciales como *C♯* y *Java* cuyas nuevas versiones agregan nuevas construcciones sintácticas sobre las versiones originales.

Referencias

[1] Waga: Proceedings of the international conference on attribute grammars and their applications, 1990.

[2] Alblas, H., and Melichar, B., Eds. *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings* (1991), vol. 545 of *Lecture Notes in Computer Science*, Springer.

[3] Bass, L. *Software architecture in practice*. Addison-Wesley, Boston, USA, 2003.

[4] Boyland, J. Remote attribute grammars. *JACM 52*, 4 (July 2005), 627–687.

[5] de Freitas, A. V., and Neto, J. J. Conception of adaptive programming languages. In *MS'06: Proceedings of the 17th IASTED international conference on Modelling and simulation* (Anaheim, CA, USA, 2006), ACTA Press, pp. 453–458.

[6] Ekman, T. *Extensible Compiler Construction*. PhD thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 2006. Dissertation 25.

[7] Garlan, D., and Shaw, M. An introduction to software architecture. Tech. Rep. CMU-CS-94-166, Carnegie Mellon University, January 1994.

[8] Gough, J. *Compiling for the .NET Common Language Runtime*. Prentice Hall PTR, 2001.

[9] Group, T. V. Xml proposal for automata description. Tech. rep., EPITA Research and Development Laboratory, 2006.

[10] Hedin, G. Reference attributed grammars. *Informatica (Slovenia) 24*, 3 (2000).

[11] III, W. H. M. *Incremental Static Semantic Analysis*. PhD thesis, EECS Department, University of California, Berkeley, 1997.

[12] Iwai, M. K., and Neto, J. J. Introdução às gramáticas adaptativas. Tech. rep., Escola Politécnica da USP, São Paulo, Brasil, 2000.

[13] Knuth, D. E. Semantics of context-free languages. *Mathematical Systems Theory 2*, 2 (1968), 127–145.

[14] Knuth, D. E. The genesis of attribute grammars. In *WAGA: Proceedings of the international conference on Attribute grammars and their applications* (New York, NY, USA, 1990), Springer-Verlag New York, Inc., pp. 1–12.

[15] Magnusson, E., and Hedin, G. Circular reference attributed grammars - their evaluation and applications. *Electr. Notes Theor. Comput. Sci. 82*, 3 (2003).

[16] Mariani, R. Garbage collector basics and performance hints. Tech. rep., .NET Development (General) Technical Articles, April 2003.

[17] Mernik, M., c, M. L., cau˘ sević, E. A., and ˘ Zumer, V. Multiple Attribute Grammar Inheritance. In *Second Workshop on Attribute Grammars and their Applications, WAGA'99* (Amsterdam, The Netherlands, 1999), D. Parigot and M. Mernik, Eds., INRIA rocquencourt, pp. 57–76.

[18] Neto, J. J., and Magalhães, M. E. S. Reconhecedores sintáticos - uma alternativa didática para uso em cursos de engenharia. *XIV Congresso Nacional de Informática* (1981), 171–181.

[19] Paakki, J. Attribute grammar paradigms – a high-level methodology in language implementation. *ACM Comput. Surv. 27*, 2 (1995), 196–255.

[20] Pagan, F. G. *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice Hall, 1981.

[21] Richter, J. Garbage collection: Automatic memory management in the microsoft .net framework. *MSDN Magazine* (November 2000).

[22] Sommerville, I. *Ingeniería del Software*. Pearson Addison-Wesley, Madrid, España, 2005.

[23] Wirth, N. *Compiler Construction*. Addison-Wesley, 1996.

[24] Wonnacott, D. G. Attribute grammars and the teaching of compiler design and implementation. *J. Comput. Small Coll. 22*, 3 (2007), 121–127.