BASES PARA UN LENGUAJE DE DESCRIPCION DE LA SEMANTICA DE LOS LENGUAJES DE PROGRAMACION

A. Campos, Y. Eterovic, G. García, R. Santis
Dpto. de Ciencia de la Computación
P. Universidad Católica de Chile
Casilla 6177 - Correo 22
Santiago - CHILE

RESUMEN

Se presentan las bases para un lenguaje que permite describir la semántica de los lenguajes de programación. El esquema propuesto está orientado, principalmente, a facilitar la producción automática de analizadores semánticos, que puedan ser usados en compiladores para los lenguajes descritos con este mecanismo. Para estos efectos se considera que el proceso de generación de código intermedio es parte integrante de la fase de análisis semántico de los compiladores. El esquema de descripción resultante, que se basa en el empleo de sintaxis abstracta para su definición, es claramente operacional. Se da especial énfasis a facilitar la descripción de las características más comunes en los lenguajes de programación de mayor uso en la actualidad. El lenguaje de descripción provee herramientas para el control de flujo, la definición de dominios semánticos, el manejo de la tabla de símbolos, y la generación de código intermedio.

1.- Introducción

En el último tiempo, la construcción de compiladores se ha visto facilitada por la aparición de diversas herramientas computacionales, que permiten generar en forma automática los módulos que implementan algunas de sus fases. Esto ha permitido que la tarea de escribir un compilador completo se haya reducido a escribir la entrada a estos programas, y a escribir aquellas partes que no son generadas en forma automática.

Actualmente existen generadores para las fases de análisis léxico, como Lex (Lesk, 1975), y de análisis sintáctico, por ejemplo Yacc (Johnson, 1975). Estos generadores y sus similares son programas que, a partir de una descripción formal de las características léxicas y sintácticas del lenguaje de programación, producen un módulo para la fase respectiva de un compilador de ese lenguaje.

También se han construido algunos generadores automáticos de generadores de código, por ejemplo el descrito por Glanville y Graham (1978), que a partir de una especificación de las características de la máquina objeto producen un generador de código para esa máquina. Por otra parte, el proceso de producción de analizadores semánticos ha resultado más resistente a la automatización, aunque también se han hecho esfuerzos importantes en esa dirección (Sethi, 1983).

En este artículo se presentan las bases de un lenguaje para describir la semántica de los lenguajes de programación. El mecanismo propuesto está orientado, principalmente, a que estas descripciones permitan generar, en forma automática, analizadores semánticos para los compiladores de esos lenguajes. Con este objetivo práctico en mente, pierde relevancia la descripción de los conceptos y entes semánticos en sí misma, y adquiere mucho más importancia el tener mecanismos conceptualmente claros y de fácil aplicación a los lenguajes de programación. El esquema resultante para el lenguaje de descripción de la semántica presenta un enfoque claramente operacional.

La definición de este lenguaje es parte de un proyecto para desarrollar un ambiente computacional que proveerá herramientas que permitan automatizar la producción de compiladores (Campos y Eterovic, 1988). En especial, se desea que este sistema facilite la descripción de aquellas características más comunes entre los lenguajes más usados en la actualidad. El lenguaje ha sido definido para ser usado como el medio a través del cual se describirán, en este ambiente, las fases de análisis semántico y de generación de código intermedio de los compiladores.

2.- Análisis Semántico

Durante la fase de análisis semántico, el compilador debe recoger y almacenar toda la información necesaria sobre los objetos que emplea el programa. Esta información puede estar explícita o implícita en el código fuente, dependiendo del tipo de información y del lenguaje de programación. La estructura de datos usada para este fin se conoce con el nombre de tabla de símbolos. Sin embargo, ella no es necesariamente una tabla propiamente tal, puede ser cualquier estructura de datos apropiada, incluso una base de datos (Horwitz y Teitelbaum, 1986). La información almacenada en esta estructura es

usada posteriormente para realizar las verificaciones semánticas, y también para la generación de código.

Las verificaciones semánticas incluyen todas aquellas reglas del lenguaje que no pueden ser impuestas mediante la sintaxis. Su objetivo es detectar el uso de construcciones que a la luz de la definición del lenguaje de programación producen programas que, aunque gramaticalmente correctos, no tienen un significado definido. Importante entre ellas, es la revisión de la compatibilidad de tipo entre los objetos usados en el programa, y la consiguiente determinación de las coerciones que puedan ser necesarias.

Se ha estimado conveniente incluir en esta misma etapa la generación de código intermedio. Desde un punto de vista conceptual, el código a ser generado es, obviamente, una definición del significado del programa, es decir, de su semántica. Por otra parte, desde el punto de vista de la implementación, ambos procesos pueden definirse y realizarse en forma simultánea, evitando la necesidad de construir explícitamente una nueva representación intermedia del programa completo. El código intermedio generado por todos los compiladores producidos en este ambiente ha sido definido como el lenguaje de ensamble de una máquina de stack hipotética (Campos y otros, 1987). Esta máquina virtual sirve entonces de base para la definición de ciertos aspectos del lenguaje de descripción de la semántica.

3.- Sintaxis Abstracta

En el proceso de compilación, el análisis semántico se efectúa a continuación del análisis sintáctico. La entrada a esa fase consiste en una representación del programa que se está compilando, en la forma de un árbol sintáctico abstracto que es producido por el analizador sintáctico. Este árbol es una versión condensada del árbol de derivación que representa al proceso de análisis sintáctico (Aho y Ullman, 1977).

En este proceso previo, parte importante de los símbolos terminales presentes en el código fuente original pueden ser dejados fuera del árbol. En particular, todos aquéllos que proveen el azúcar sintáctica del lenguaje: la estructura del programa ya no está implícita en la secuencia lineal de símbolos usados en el programa fuente, sino que está explícitamente caracterizada por la estructura del árbol producido. Por otra parte, es usual que el analizador sintáctico haya resuelto también algunos problemas de índole semántica, por ejemplo aquéllos relativos a la asociatividad y prioridad de los operadores en una expresión, características que también quedan reflejadas en la estructura del árbol.

También es usual que al describir la sintaxis de un lenguaje de programación, y en particular al definir las categorías sintácticas, se tenga en consideración su semántica. Esto hace posible que una parte del trabajo

semántico pueda ser realizada a nivel sintáctico. Por todas estas razones resulta adecuado definir la semántica del lenguaje a partir de la definición de sus estructuras sintácticas. Sin embargo, no parece razonable comenzar exactamente desde el mismo punto, sino que es preferible comenzar desde algo mucho más simple: la sintaxis abstracta.

La sintaxis abstracta es una gramática libre de contexto que en sus producciones sólo contiene variables y símbolos terminales semánticamente significativos. Ella debe describir un lenguaje que incluya a aquél cuyos árboles de derivación son los árboles sintácticos abstractos producidos por el analizador sintáctico. Es decir, debe describir cada árbol que pueda ser producido durante el análisis sintáctico de un programa, pero puede describir, además, algunos árboles que nunca serán producidos debido a ciertas restricciones sintácticas del lenguaje de programación. Incluso, es perfectamente posible que sea una gramática ambigua, pues el árbol ya habrá sido construido durante el análisis sintáctico.

La sintaxis abstracta se emplea como base del lenguaje de descripción de la semántica. A su vez, el lenguaje empleado para definir las producciones de la sintaxis abstracta es muy simple y posee los siguientes metasímbolos:

=>	denota la producción.
%l(...)	denota una lista de lo que aparece entre paréntesis.
[...]	denota el nombre o número de una producción.

También es posible emplear comentarios que clarifiquen las producciones. Cualquier secuencia de carácteres que aparezca entre apóstrofes ('), o comillas ("), es considerada un comentario e ignorada como parte de la definición de la sintaxis abstracta. Así, por ejemplo, se puede definir:

```
[1]  A => 'program' %l(B) C
[2]  B => Identificador
[3]  C => 'begin' %l(Sentencia) 'end'
```

Esta definición indica que una lista de Identificador seguida de una lista de Sentencia es derivable desde A. Nótese que tanto "A", "B" y "C", como también "Sentencia" son variables de la gramática, y que "Identificador" representa, supuestamente, a un terminal en la definición sintáctica. Por otra parte, "program", "begin" y "end", que seguramente son símbolos terminales en la definición sintáctica, sólo aparecen como comentarios en esta definición, y su único objetivo es servir de ayuda mnemotécnica en su interpretación.

4.- Lenguaje de Descripción de la Semántica

La sintaxis abstracta sirve de punto de partida para describir la semántica de los lenguajes de programación. Para ello, a cada producción en

esta gramática se le asocian varios atributos. A su vez, estos atributos son empleados en la definición de otros atributos, asociados a otras o a la misma producción. Los atributos son, en realidad, rutinas o variables que representan acciones o valores, a través de los cuales se puede definir la transferencia de información entre los distintos nodos del árbol sintáctico abstracto. Mencionarlos en la definición de otro atributo equivale a especificar una llamada a la rutina, o el empleo del valor asociado a la variable.

Durante el análisis semántico, a través de las producciones, los atributos quedan asociados a los nodos correspondientes del árbol sintáctico abstracto del programa que se está compilando. El atributo que debe usarse en cada caso queda determinado en forma única por la estructura específica de este árbol. Cuando se requiere un atributo, ya sea una acción o una variable, para un nodo del árbol, se emplea la definición dada en la producción que corresponde a ese nodo. Es decir, aquélla en que la variable del lado izquierdo corresponde al nodo, y en que los símbolos del lado derecho corresponden exactamente a sus hijos.

El proceso que realiza el analizador semántico corresponde entonces a un recorrido sobre el árbol sintáctico abstracto, en que un mismo nodo podría ser visitado varias veces. Durante este recorrido, se realizan las acciones correspondientes a los atributos que sean requeridos por otros atributos. La forma exacta de este recorrido queda implícita en la descripción semántica al requerirse los distintos atributos definidos y, en general, será diferente para cada programa que se compile. Los atributos que son variables han sido introducidos, precisamente, como una herramienta que permite evitar que un mismo nodo sea visitado demasiadas veces. Ellos pueden almacenar valores calculados en una visita hasta cuando sean requeridos realmente, sin que sea necesario visitar una parte del árbol por segunda vez.

El lenguaje de descripción propiamente tal, que permite especificar los atributos, posee una serie de herramientas orientadas a facilitar la implementación de las características más comunes en los lenguajes de programación actuales. Estas herramientas no son rígidas, permitiendo que una misma tarea pueda ser definida de varias formas, con lo que se obtiene la ductilidad necesaria para acomodar las posibles variaciones. En particular, provee estructuras de control de flujo para una adecuada codificación de los atributos, permite la utilización de variables locales con el objetivo de disminuir el nivel de acoplamiento entre ellos, y permite la especificación y uso de otras rutinas y macros para facilitar su definición.

Por ejemplo, a una producción de la sintaxis abstracta para la suma de dos expresiones, se le podría definir el atributo "codigo" como sigue:

```
suma => exp '+' exp
     -codigo: begin
              exp_1.codigo
              exp_2.codigo
              Suma(exp_1.tipo, exp_2.tipo)
     end.
```

Este atributo representa una acción. Su evaluación se efectúa evaluando primero el atributo "codigo" de la primera expresión. A continuación se hace algo similar con la segunda expresión. Finalmente, se llama a la rutina o macro "Suma", pasando como parámetros los tipos de esas expresiones. Los tipos son también atributos de las producciones para las expresiones. Como ya se ha recorrido el sub-árbol correspondiente a cada expresión para evaluar el atributo "codigo", es posible que también se conozca el tipo resultante para cada una de ellas y que no sea necesario hacer un nuevo recorrido sobre esas partes del árbol.

Considerando que un objetivo muy importante de las descripciones semánticas hechas con este lenguaje es la generación automática de analizadores semánticos, también se han incluido herramientas para simplificar la definición de los dominios semánticos, el manejo de la tabla de símbolos, y la especificación del proceso de generación de código intermedio.

5.- Dominios Semánticos

Con el objetivo de formalizar los dominios semánticos, a la vez que clasificar los distintos entes que puedan aparecer en la descripción semántica de un lenguaje de programación, el lenguaje de definición dispone de algunos dominios predefinidos y provee mecanismos para la especificación formal de otros.

El lenguaje predefine los siguientes tipos de datos: INTEGER, REAL, BYTE, WORD y LABEL. Los cuatro primeros corresponden a tipos de datos presentes en prácticamente cualquier máquina real, y son equivalentes a algunos de los tipos básicos definidos en la máquina virtual cuyo código será generado. Por otra parte, el tipo de dato LABEL corresponde a nombres para direcciones de memoria, los que serán convertidos a la dirección absoluta real con posterioridad. Además, eventualmente, todos aquellos tipos de datos básicos que aparecen definidos en la máquina hipotética serán dominios predefinidos.

Para especificar nuevos dominios, a partir de aquéllos definidos previamente, el lenguaje provee las operaciones de unión y producto cartesiano, y también el concepto de función. Las definiciones pueden emplear, en forma recursiva, los dominios que están siendo definidos. Así, por

ejemplo, las siguientes son definiciones válidas para algunos tipos de datos posibles en un lenguaje de programación:

entero = INTEGER
doble = REAL × REAL
elemento = entero + REAL + doble
arreglo = INTEGER -> elemento
lista = (elemento × lista) + {λ}

En estas definiciones, el símbolo "+" representa la unión, "×" representa el producto cartesiano, y "->" representa la idea de función. El símbolo "λ" se usa para denominar a una lista en particular, que corresponde al concepto de la lista vacía o nula. Los símbolos "{" y "}" encierran un conjunto descrito en forma literal. Por último, el símbolo "=" representa la definición de un dominio, y los símbolos "(" y ")" se usan sólo para agrupar símbolos.

Estos mecanismos permiten definir los dominios básicos del lenguaje de programación cuya semántica se está describiendo. Por ejemplo, para Pascal se definirán los arreglos, los registros, etcétera. Al respecto, es importante destacar que no debe confundirse la definición de los dominios semánticos con la declaración, en un programa, de algunas instancias de él.

El objetivo principal de estas especificaciones es aclarar la relación conceptual entre los tipos de la máquina virtual, y los tipos propios del lenguaje que se describe. Con ellas, se obtiene también una manera precisa de describir las operaciones básicas de los tipos de este lenguaje; lo que permite, por ejemplo, definir las funciones de coerción, suma, o cualquier otra aplicable a un tipo de datos, o conjunto de ellos.

6.- Tabla de Símbolos

Se ha establecido un modelo general de tabla de símbolos de manera que sea fácil definir las funciones para administrarla en el caso de los lenguajes de programación de mayor uso en la actualidad, pero permitiendo también especificaciones diferentes. Para posibilitar la implementación de las distintas reglas de visibilidad de nombres presentes en los lenguajes de programación, la tabla de símbolos no existe como un todo único, sino que como un conjunto de tablas locales. Este conjunto se organiza como un grafo, usualmente un árbol, en que las tablas locales son los vértices. Este esquema permite definir en forma simple la regla de visibilidad de lenguajes como, por ejemplo, Pascal, en que un objeto se busca primero en una tabla local y, si no se encuentra allí, se busca en su ancestro dentro del árbol.

Para ciertos lenguajes de programación, el grafo puede simplificarse a un stack de tablas locales. Es por esto que en el lenguaje de descripción de la

semántica se proveen facilidades para definirlos y manipularlos. Específicamente, se predefinen las funciones para poner (Push) y sacar (Pop) una tabla local del stack, además de las funciones básicas para crear las tablas locales y los stacks. Concretamente, existen las siguientes operaciones predefinidas:

CreaTabla(t: Tabla)
CreaStack(s: Stack)
Push(t: Tabla, s: Stack)
Pop(s: Stack)

Los objetos almacenados en estas tablas locales se denominan símbolos. En general, ellos representan a los diferentes entes usados en un programa sobre los cuales se desea mantener alguna información. En este modelo, un símbolo consta de un nombre que lo identifica y de una serie de atributos. A su vez, los atributos son tríos compuestos por su nombre, tipo y valor. El nombre de cada atributo es único para un símbolo dado, y sirve para identificarlo. Los nombres del símbolo y de los atributos son valores de tipo IDENTIFICADOR. El valor de un atributo puede ser un valor numérico, una secuencia de carácteres, o puede ser un puntero a otro símbolo, a una tabla local, o a un trozo de código.

El lenguaje provee funciones primitivas que permiten crear y buscar un símbolo en una tabla local, agregar y eliminar un atributo a un símbolo, y modificar el valor de un atributo:

CreaSimbolo(n: IDENTIFICADOR, t: Tabla): Simbolo
BuscaSimbolo(n: IDENTIFICADOR, t: Tabla): Simbolo
AgregaAtributo(s: Simbolo, n: IDENTIFICADOR, t: Tipos, v: Tipo)
BorraAtributo(s: Simbolo, n: IDENTIFICADOR)
ModificaAtributo(s: Simbolo, n: IDENTIFICADOR, v: Tipo)

Como herramienta para facilitar el uso de los stacks de tablas locales, también se predefine una función que busca un símbolo dentro de un stack:

BuscaEnStack(n: IDENTIFICADOR, s: Stack): Simbolo

Dado un nombre y el stack de tablas, esta función busca un símbolo con ese nombre. La búsqueda se hace en cada tabla local, comenzando desde la que está en el tope del stack, y siguiendo ordenadamente hacia abajo. Devuelve el primer símbolo que encuentra con ese nombre, o un símbolo nulo en caso de que no existiera uno.

Si se quisiera implementar otro tipo de búsqueda, o para tablas de símbolos en que las tablas locales están arregladas de otra forma, se deberá programar la función correspondiente. Para ello se puede emplear como base

la función de búsqueda en una tabla local (BuscaSimbolo) y un número de punteros a otras tablas.

Otra facilidad provista en el lenguaje de descripción semántica es una función generadora de símbolos internos. Esta función crea nombres únicos para símbolos, garantizando que será diferente al de cualquier otro símbolo generado por ella, o al nombre de cualquier objeto que aparezca en el programa. Esto permite incluir símbolos interno creados por el analizador semántico y que no provienen, necesariamente, de objetos del programa.

7.- Generación de Código Intermedio

Como se dijo anteriormente, a cada producción de la sintaxis abstracta se le definen atributos que, durante el análisis semántico, quedan asociados al nodo correspondiente en el árbol sintáctico abstracto. Estos atributos pueden ser rutinas que describen acciones, y cuyos parámetros son los hijos del nodo respectivo. El paso de estos parámetros queda implícito en la definición del atributo.

Este mismo mecanismo es el que debe usarse para especificar el proceso de generación de código intermedio. Si se requiere la generación de ciertas instrucciones, debe haber un atributo que lo haga. El lenguaje de descripción semántica provee, además, funciones que permiten emitir las instrucciones de la máquina virtual a un archivo, desde donde pueden ser posteriormente optimizadas, interpretadas o traducidas a código de una máquina objeto real. En caso que se usara otro conjunto de instrucciones como lenguaje intermedio para los compiladores, estas rutinas pueden redefinirse con facilidad.

En el ejemplo del atributo "codigo", definido en la producción para la suma de dos expresiones y presentado anteriormente, se puede suponer que él genera las instrucciones necesarias para evaluar la suma. En ese caso, el atributo "codigo" de las expresiones debe emitir las instrucciones para dejar su valor en el tope del stack de la máquina hipotética. Esto puede hacerlo directamente, o a través de atributos definidos para las variables que aparezcan al lado derecho de producciones para las expresiones. Con posterioridad, la rutina o macro "Suma" emitirá, de acuerdo al tipo de ellos, las instrucciones para sumar dos valores que están en el tope del stack.

Es importante destacar que esa definición es independiente de si se genera código o no. Por ejemplo, también se podría interpretar como una especificación del valor de una suma. En ese caso, el atributo "codigo" representaría el proceso de evaluación de una suma o de una expresión. Este proceso dejaría los valores calculados en el tope de un stack de evaluación, y la rutina o macro "Suma" tomaría esos valores y produciría el resultado final.

8.- Conclusiones

Se han presentado las bases para la definición de un lenguaje que permite describir la semántica de los lenguajes de programación, cuyo objetivo principal es el de facilitar la producción de un analizador semántico a partir de esas descripciones. El resultado es un lenguaje de corte netamente operacional, que incluye algunas facilidades para definir con simplicidad las características más usuales entre los lenguajes de mayor difusión en la actualidad. Sin embargo, también posee la flexibilidad suficiente para acomodar alternativas posibles de presentarse en los lenguajes de programación. Se ha dado especial importancia a características que permitan definir con comodidad los tipos involucrados, las acciones a efectuar por el administrador de la tabla de símbolos, y el proceso de generación de código intermedio.

AGRADECIMIENTOS

Este trabajo es parte del proyecto DIUC 19/86: "Desarrollo de un Ambiente para la Generación de Compiladores", que ha sido financiado por la Dirección de Investigación de la Pontificia Universidad Católica de Chile, organismo al cual los autores desean expresar sus más sinceros agradecimientos.

REFERENCIAS

1.- Aho, A.V. y Ullman, J.D. (1977) Principles of Compiler Design. Addison-Wesley, Reading.

2.- Campos, A., Eterovic, Y. (1988) A Compiler Production Environment. Presentado para consideración a The Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, Enero 1988.

3.- Campos, A., Eterovic, Y., García, G., Santis, R. (1987) Implementación de un Computador Virtual con Capacidad de Multiprocesamiento. Aceptado en el Octavo Congreso Latinoamericano e Ibérico sobre Métodos Computacionales para Ingeniería. Rio de Janeiro, Noviembre 1987.

4.- Glanville, R.S. y Graham, S.L. (1978) A New Method For Compiler Code Generation. Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages, New York, Enero 1978, 231-240.

5.- Horwitz S. y Teitelbaum, T. (1986) Generating Editing Environments Based on Relations and Attributes. ACM Transactions on Programming Languages and Systems, Vol 8, Nº4, 577-608.

6.- Johnson, S.C. (1975) Yacc: Yet Another Compiler-Compiler. CSTR 32, Bell Laboratories, Murray Hill.

7.- Lesk, M.E. (1975) LEX - A Lexical Analyzer Generator. CSTR 39, Bell Laboratories, Murray Hill.

8.- Sethi, R. (1983) Control Flow Aspects of Semantic-Directed Compiling. ACM Transactions on Programming Languages and Systems, Vol 5, Nº4, 554-595.