
LENGUAJES Y COMPILADORES - APUNTES DE CLASE

Estos apuntes de clase son una reedición de los apuntes diseñados por Daniel Fridlender en el ciclo 2010.

Profesores: Héctor Gramaglia, Miguel Pagano, Carlos Bederián
Lista de Mail: famaf-compiladores@googlegroups.com
Suscribirse en groups.google.com/group/famaf-compiladores.

Régimen de promoción, regularidad y aprobación

Parciales: 3 - 21/4 - 19/5 - 16/6
Promoción: obteniendo al menos 7 en cada uno de los parciales y aprobando el taller.
Regularidad: aprobando 2 parciales y aprobando el taller.
Taller: se implementarán intérpretes y compiladores de lenguajes de programación.
Examen: examen con parte escrita y parte oral.
Alumnos libres: examen con ejercicios/preguntas adicionales + defensa del taller.

Bibliografía: Reynolds, Theories of Programming Languages, Cambridge University Press, 1998.

SEMÁNTICA DE LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación son un ejemplo fructífero para mostrar la manera en que diversas formas de atribuir significados son posibles, haciendo cada una de ellas un aporte sustantivo a la comprensión y utilidad del mismo. Los mismos muestran la necesidad de trascender la modalidad estática del significado vigente en la lógica formal y la matemática (denotación), y apelar al sentido finito y dinámico, distinción que ya fuera hecha por Frege en el contexto de la lógica formal.

Podemos destacar cuatro maneras diferentes formas de dar significado a los programas (o al lenguaje). Las primeras tres de ellas ponen énfasis en el sentido dinámico finito, en tanto la cuarta se refiere a un significado ideal en el universo formado por los objetos matemáticos.

  informal, intuitiva: explica el funcionamiento intuitivo de los programas
               (ejemplo: manuales, documentación tipo javadoc)
  axiomática: establece cómo razonar sobre programas (ejemplo: {P / v <- e} v:= e {P})
  operacional: describe cómo se ejecuta un programa (ejemplo: intérprete)
  denotacional: mapea programas a su significado (ejemplo: compilador)

A partir del desarrollo de la Teoría de Dominios la semántica denotacional adquiere un relevancia especial, no sólo por tratarse de objetos matemáticos perfectamente definidos en el contexto de una teoría particular, sino además porque comienza a ser utilizada como "la definición" del lenguaje y luego, si se proponen otras semánticas (operacional, axiomática), se las demuestra correctas con respecto a dicha definición.

EJEMPLO: para ejemplificar tomamos el siguiente lenguaje, a pesar de que no es un lenguaje de programación, es un mini-lenguaje de expresiones enteras: constantes no negativas, menos unario y más y por binarios.

```
<intexp> ::= 0 | 1 | 2 | ...
           | -<intexp> | <intexp> + <intexp> | <intexp> * <intexp>
```

Antes de ilustrar las diferentes maneras de dar la semántica, una "discusión sobre sintaxis": ¿en qué sentido esta gramática describe la sintaxis del lenguaje?

Observemos ejemplos de frases generadas por esta gramática:

```
142
-15
-15+3
2+3+4
2*-4
```

La gramátíca es ambigua: algunas frases admiten diferentes maneras de generarse. Por ejemplo, 2+3+4:

```
<intexp> -> <intexp> + <intexp> -> 2 + <intexp> -> 2 + <intexp> + <intexp> -> .. -> 2 + 3 + 4
<intexp> -> <intexp> + <intexp> -> <intexp> + 4 -> <intexp> + <intexp> + 4 -> .. -> 2 + 3 + 4
```

La primera manera de generarse dicha frase, se corresponde intuitivamente con asociar a derecha (es decir, con 2 + (3 + 4)) y la segunda con asociar a izquierda (es decir, con (2 + 3) + 4). Pero ninguna de estas dos frases puede generarse porque los paréntesis no están entre los símbolos terminales de la gramática.
--
Pregunta: ¿con cuáles de las otras frases mencionadas más arriba ocurre lo mismo? Desarrollar.
--
El problema podría resolverse agregando paréntesis y desambiguando la gramática. Pero eso la haría bastante más complicada. Y peor aún, ocultaría algo que en la gramática de arriba es evidente: que el mini-lenguaje tiene constantes no negativas, un operador unario (-) y dos operadores binarios (+ y *). Por esta razón, se prefiere la gramática que se acaba de dar.

Pero debemos dejar sentado, justamente por el problema de ambigüedad, que la gramática no pretende describir la notación exacta o concreta, sólo la estructura de las frases, dice cuáles son las construcciones que hay en el lenguaje. No interesa cómo se escriben las frases sino qué frases hay. Diremos que es una "gramática abstracta", que describe la "sintaxis abstracta" y determina "frases abstractas". Trabajar a este nivel de abstracción es muy conveniente ya que nos permite desentendernos de detalles propios de la notación concreta que no tienen interés cuando se trata de dar significado a las frases.

El libro de Reynolds hace un tratamiento muy detallado del significado preciso de gramática abstracta. Es muy interesante y lectura recomendada para entender esto con precisión. Pero también es difícil y no es imprescindible para comprender la materia.

En las pocas ocasiones en que nos resulte necesario precisar notación específica resolveremos los problemas de ambigüedad mencionados estableciendo precedencias, "stopping symbols" y utilizando los paréntesis que sean necesarios.

Con esto finalizamos la "discusión sobre sintaxis" iniciada apenas presentamos la gramática del mini-lenguaje. A continuación ilustramos la manera en la que procederemos para dar semántica denotacional.

Sse define un dominio semántico, Z, y un mapeo [[]] de <intexp> en dicho dominio semántico:

```
   [[]] ∈ <intexp> -> Z

    [[0]] = 0 ...
    [[-e]] = - [[e]]
    [[e+f]] = [[e]] + [[f]]
    [[e*f]] = [[e]] * [[f]]
```

Estudiaremos varios lenguajes siguiendo este último patrón:
  1) presentación de la gramática abstracta
  2) definición de los dominios sintácticos
  3) presentación de las ecuaciones que determinan la semántica denotacional

Observemos las ecuaciones que obtuvimos para este mini-lenguaje de expresiones:

```
    [[0]] = 0 ...
    [[-e]] = - [[e]]
    [[e+f]] = [[e]] + [[f]]
    [[e*f]] = [[e]] * [[f]]
```

Estas ecuaciones dan (un único) significado a las frases del mini-lenguaje. Por ejemplo

```
    [[5+2]] = [[5]] + [[2]]
            = 5 + 2
            = 7
    [[3*(5+2)]] = [[3]] * [[5+2]]
                = 3 * 7
                = 21
    [[3*(5+2)+(-2)]] = [[3*(5+2)]] + [[-2]]
```

```
                    = 21 + (- [[2]])
                    = 21 - 2
                    = 19
```

LENGUAJE Y METALENGUAJE

Observemos nuevamente las ecuaciones que obtuvimos para este mini-lenguaje de expresiones:

```
    [[0]] = 0 ...
    [[-e]] = - [[e]]
    [[e+f]] = [[e]] + [[f]]
    [[e*f]] = [[e]] * [[f]]
```

La presentación utiliza 2 lenguajes: el mini-lenguaje de expresiones y el lenguaje EN que hicimos la definición. Al primero se lo llama "lenguaje" y al segundo "metalenguaje":

En [[0]] = 0, el primer 0 es del lenguaje (es la frase 0) y el segundo 0 es del metalenguaje (es el número entero 0).

En [[-e]] = - [[e]], el primer - es del lenguaje (es el operador unario -) y el segundo - es del metalenguaje (es la función que devuelve el opuesto de su argumento).

Similarmente en las otras dos ecuaciones.

Cada una de las tres últimas ecuaciones representa, en realidad, infinitas ecuaciones. Por ejemplo, [[-e]] = - [[e]] establece una propiedad que vale cualquiera sea la expresión e. Pero e NO es una expresión, sólo es un objeto que representa cualquier expresión del lenguaje. Es una variable. Pero NO es una variable del lenguaje (revisemos la gramática para comprobar que no hay variables en el lenguaje, sólo constantes y operadores). Es una variable del metalenguaje que utilizamos para escribir una ecuación como [[-e]] = - [[e]] en vez de infinitas ecuaciones, una para cada expresión. A estas variables del metalenguaje se las llama "metavariables".

La primer ecuación, en cambio, fue escrita sólo para la frase 0, y los puntos suspensivos expresan que hay una ecuación como esa para cada número natural. Podríamos escribir [[n]] = n para cada n en N. Pero acá estaríamos abusando de la notación, ya que el primer n es una frase y el segundo es un número natural. La metavariable n no representa en ambos lugares exactamente lo mismo. Esto suele resolverse escribiendo [[ñ]] = n para cada n en N, donde ñ es la manera de escribir el número natural n en el lenguaje.

METACIRCULARIDAD

Volvamos a observar las ecuaciones que obtuvimos para este mini-lenguaje de expresiones:

```
    [[ñ]] = n
    [[-e]] = - [[e]]
    [[e+f]] = [[e]] + [[f]]
    [[e*f]] = [[e]] * [[f]]
```

Las ecuaciones no parecen decir mucho, definen la semántica del operador del lenguaje - (resp +, *) en función de la función del metalenguaje - (resp +, *). Como ya observamos, no hay circularidad en la definición ya que en un caso se trata de un operador del lenguaje y en otro del metalenguaje. De todas formas, esta "aparente" circularidad tiene un nombre: "metacircularidad".

La metacircularidad en algunas ecuaciones es habitual al definir semántica. Hay que estar atentos ya que se corre el riesgo de llevar inconscientemente "vicios" del metalenguaje al lenguaje que se intenta definir.

SOBRE EL METALENGUAJE

El metalenguaje que se utilizará a todo lo largo de la materia es la teoría de conjuntos habitual de la matemática. Tiene importantes similitudes con Haskell, por ello muchas de las definiciones son fácilmente transcribibles en ese lenguaje de programación.

TAREA PARA LA PRÓXIMA CLASE A LAS 8:59 a.m.:

Implementar en Haskell un evaluador del mini-lenguaje. Para ello, definir un tipo de datos para las expresiones y una función de evaluación. No implementar el parser. Ejemplificar evaluando dicha función en diferentes expresiones.

Es importante remarcar que Haskell no es idéntico al lenguaje de la matemática que utilizamos en clase. Quizá la diferencia más importante es que en el lenguaje de la matemática que se utilizará, las funciones son siempre totales, en cambio en Haskell hay funciones parciales:

```
fact :: Z -> Z
fact n = if n == 0 then 1 else n * fact (n-1)
```

fact no está definida para los números negativos

```
head :: [a] -> [a]
head (a:as) = a
```

head no está definida para las listas vacías

```
bottom :: a -> b
bottom a = bottom a
```

bottom no está definida para ningún argumento

SIGNIFICADO ÚNICO

Volvamos a observar las ecuaciones que obtuvimos para este mini-lenguaje de expresiones:

```
    [[ñ]] = n
    [[-e]] = - [[e]]
    [[e+f]] = [[e]] + [[f]]
    [[e*f]] = [[e]] * [[f]]
```

Estas ecuaciones dan significado único a las frases del lenguaje. Se puede demostrar (fuera del alcance de este curso) que si se respetan ciertos criterios sintácticos para las ecuaciones, seguro que hay significado único. Cuando hablamos de ecuaciones "semánticas" estamos dando a entender que siguen algún criterio que garantiza existencia y unicidad.

Un conjunto de ecuaciones es "dirigido por sintaxis" cuando se satisfacen las siguientes condiciones:
  1) hay 1 ecuación por cada producción de la gramática abstractat
  2) cada ecuación que expresa el significado de una frase compuesta, lo hace puramente en función de los significados de sus subfrases inmediatas

dirección por sintaxis => existencia y unicidad del significado

Se dice que una semántica es "composicional", cuando el significado de una frase no depende de ninguna propiedad de sus subfrases, salvo de sus significados.

Composicionalidad es muy importante ya que implica que podemos reemplazar una subfrase f de e por otra de igual significado que f sin alterar el significado de la frase e.

Composicionalidad no es lo mismo que dirección por sintaxis: composicionalidad habla de una propiedad de la semántica, mientras que dirección or sintaxis habla de la forma en que se definió dicha semántica.

dirección por sintaxis => composicionalidad

Concluyendo, dirección por sintaxis garantiza existencia y unicidad del significado y también garantiza composicionalidad, todas propiedades deseables. Por ello, insistiremos en que nuestras ecuaciones sean dirigidas por sintaxis.