Lenguajes de Programación 2020-I[*]
Nota de clase 4, Sintaxis[**]

Favio E. Miranda Perea Lourdes Del Carmen González Huesca
Facultad de Ciencias UNAM

19 de agosto de 2019

En esta nota desarrollamos algunos aspectos de la sintaxis de un lenguaje. Para esto nos serviremos de dos clases de objetos, las cadenas de texto y los árboles de sintaxis abstracta (asa). Las cadenas proporcionan una representación lineal conveniente para la interacción humana, esto se conoce como *sintaxis concreta*, y ya ha sido estudiada mediante gramáticas libres de contexto en el curso de teoría de la computación. Por otra parte los árboles de sintaxis abstracta modelan la estructura jerárquica de la sintaxis y son mucho mas adecuados que las cadenas para el análisis y la mecanización.

La sintaxis concreta de un lenguaje de programación se determina usualmente en dos partes:

- Sintaxis léxica: describe la construcción de lexemas (átomos, *tokens*, símbolos terminales). Por ejemplo: palabras reservadas, identificadores, numerales, literales, espacios, etc. La herramienta principal para descripción son las expresiones regulares.
- Sintaxis libre de contexto: describe la construcción de frases del lenguaje, por ejemplo, expresiones, definiciones, declaraciones, etc. La herramienta principal para la descripción son las gramáticas libres de contexto, por lo general en forma extendida BNF.

1. Sintaxis para un lenguaje de expresiones aritméticas

Para ejemplificar los conceptos más importantes de la sintaxis de un lenguaje utilizamos como ejemplo particular un lenguaje de expresiones aritméticas, denotado de ahora en adelante con EA, el cual está esencialmente contenido en todo lenguaje de programación real.

1.1. Sintaxis concreta de EA

La primera propuesta para sintaxis concreta es:

$$\begin{array}{lcll} E & ::= & N \mid E + E \mid E * E & \%expresiones \\ N & ::= & D \mid ND & \%numeros \\ D & ::= & 0 \mid \ldots \mid 9 & \%digitos \end{array}$$

[*]Agradecemos la colaboración de Susana Hahn Martín Lunas en la revisión 2018-I

[**]Estas notas se basan en diversas versiones del libro de Robert Harper *Practical Foundations of Programming Languages.*

Las expresiones aritméticas son aquellas generadas por la gramática siendo E el símbolo inicial.

Es bien sabido que esta gramática es ambigua, por ejemplo la expresión $27 * 4 + 3$ puede analizarse de dos formas las cuales tienen significado distinto. Esto es indeseable en un lenguaje de programación ya que genera interpretaciones ambiguas en un enunciado de programa. Como ejemplo considerese la siguiente función de evaluación que asigna a cada expresión aritmética su valor en N. Es natural usar recursión sobre las reglas que definen a la gramática de expresiones:

$$\begin{aligned} eval(0) &= 0 \\ &\vdots \\ eval(9) &= 9 \\ eval(nd) &= eval(n) * 10 + eval(d) \\ eval(e_1 + e_2) &= eval(e_1) + eval(e_2) \\ eval(e_1 * e_2) &= eval(e_1) * eval(e_2) \end{aligned}$$

Lo primero que debemos verificar es que la evaluación esté bien definida. Desgraciadamente la respuesta es no. La razón es que la ambigüedad de la gramática permite obtener una misma expresión de dos o más maneras distintas. Por ejemplo, la expresión $1 + 2 * 3$ puede evaluarse como 7 que corresponde a $1 + (2 * 3)$ o como 9 que corresponde a $(1 + 2) * 3$. De esta forma la supuesta función de evaluación no es una función en el sentido matemático.

Eliminar la ambigüedad de una gramática es un arte obscuro, no existe un algoritmo y de hecho no siempre es posible hacerlo al existir lenguajes formales con la peculiaridad de que cualquier gramática que los genere resulta ser ambigua, tales lenguajes se llaman inherentemente ambiguos[1]. Sin embargo, en algunos casos como en nuestro ejemplo, la ambigüedad puede eliminarse modelando la precedencia y asociatividad de los operadores.

Adoptamos asociatividad izquierda de los operadores de suma y producto para lo cual se jerarquiza la gramática en numeros, factores, términos y finalmente expresiones, obteniéndose la siguiente gramática:

$$\begin{array}{lcll} E & ::= & T \mid E + T & \%expresiones \\ T & ::= & F \mid T * F & \%terminos \\ F & ::= & N \mid (E) & \%factores \\ N & ::= & D \mid ND & \%numeros \\ D & ::= & 0 \mid \ldots \mid 9 & \%digitos \end{array}$$

Nuevamente el símbolo inicial de la gramática es E. Para nuestros propósitos posteriores transformamos la gramática anterior en una definición inductiva mediante las siguientes reglas de inferencia:

$$\frac{}{0\ \mathsf{D}} \cdots\cdots \frac{}{9\ \mathsf{D}}$$

$$\frac{s\ \mathsf{D}}{s\ \mathsf{N}} \qquad \frac{s_1\ \mathsf{N} \quad s_2\ \mathsf{D}}{s_1 s_2\ \mathsf{N}}$$

$$\frac{s\ \mathsf{N}}{s\ \mathsf{F}} \qquad \frac{s\ \mathsf{E}}{(s)\ \mathsf{F}}$$

[1] Un ejemplo de tal lenguaje es $\mathcal{L} = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$.

$$\frac{s\ \mathsf{F}}{s\ \mathsf{T}} \qquad \frac{s_1\ \mathsf{T} \quad s_2\ \mathsf{F}}{s_1 * s_2\ \ \mathsf{T}}$$

$$\frac{s\ \mathsf{T}}{s\ \mathsf{E}} \qquad \frac{s_1\ \mathsf{E} \quad s_2\ \mathsf{T}}{s_1 + s_2\ \ \mathsf{E}}$$

En este caso podemos probar, por inducción sobre las reglas, que la siguiente función de evaluación está bien definida:

$$\begin{aligned} eval(0) &= 0 \\ &\vdots \\ eval(9) &= 9 \\ eval(nd) &= eval(n) * 10 + eval(d) \\ eval(t + e) &= eval(t) + eval(e) \\ eval(f * t) &= eval(f) * eval(t) \\ eval((e)) &= eval(e) \end{aligned}$$

Si bien esta definición es intuitivamente clara y matemáticamente correcta, en realidad se están obviando varios detalles que son relevantes para la implementación, por ejemplo el hecho de que el dominio de la función es la unión de las cinco categorías de la gramática. Una definición más adecuada para implementarse sería separar la definición anterior en distintas funciones de evaluación, $eval_D$, $eval_N$, $eval_T$, $eval_F$ y $eval_E$, con lo cual la implementación es directa.

1.2. Sintaxis abstracta

Nuestro objetivo es razonar acerca de las expresiones aritméticas de manera inductiva, sin embargo la sintaxis concreta proporciona una definición inductiva demasiado compleja para su manejo. En su lugar podriamos usar el árbol de análisis sintáctico o el árbol de derivación de la cadena, pero esto también resulta complicado. Estas dificultades son parte del precio pagado al eliminar la ambigüedad de la gramática del lenguaje.

La sintaxis abstracta nos proporciona una representación de una expresión mediante un árbol obtenido despues de la fase de análisis sintáctico, el cual es más simple de manipular. De esta manera se evita la complejidad de la sintaxis concreta mediante el uso de operadores que determinan la forma más externa de cualquier expresión dada. En particular se evita el manejo de los paréntesis de la sintaxis concreta.

Un árbol de sintaxis abstracta (asa) es un árbol ordenado cuyos nodos están etiquetados por un operador. Cada operador tiene un índice asignado que indica el número de argumentos que recibe, los cuales corresponden al número de hijos de cualquier nodo etiquetado con él. La asignación de índices de operadores se especifica mediante una función de signatura ar definida por un conjunto de juicios de la forma $ar(o) = n$ donde o es un nombre de operador. Una vez fijo el operador de signatura, el juicio a asa se define mediante la siguiente regla:

$$\frac{ar(o) = n \quad a_1\ \mathsf{asa} \ldots a_n\ \mathsf{asa}}{o(a_1, \ldots, a_n)\ \mathsf{asa}}$$

En el caso de que $ar(o) = 0$ tenemos que la regla no tiene premisas de la forma a_i asa, por lo que cualquier nodo etiquetado con o es una hoja del árbol de sintaxis abstracta. Estos operadores utilizarán un elemento primitivo para especificar su valor, dicho elemento se define usando `[]`.

Para el caso de las expresiones aritméticas observamos que hay tres clases de expresiones que son números, sumas y productos. Estas expresiones corresponden a tres operadores $\mathtt{suma}, \mathtt{prod}$ y $\mathtt{num}$ con signatura $ar(\mathtt{suma}) = 2,\ ar(\mathtt{prod}) = 2,\ ar(\mathtt{num}) = 0$.

La sintaxis abstracta para EA se define con las siguientes reglas correspondientes a casos particulares de la regla recién definida para asa.

$$\frac{n\ \mathsf{Nat}}{\mathtt{num}[n]\ \mathsf{asa}} \qquad \frac{t_1\ \mathsf{asa} \quad t_2\ \mathsf{asa}}{\mathtt{suma}(t_1, t_2)\ \mathsf{asa}} \qquad \frac{t_1\ \mathsf{asa} \quad t_2\ \mathsf{asa}}{\mathtt{prod}(t_1, t_2)\ \mathsf{asa}}$$

Algunas veces se abusa presentando a la sintaxis abstracta con una gramática libre de contexto, por ejemplo:

$$\mathsf{asa} ::= \mathtt{num}[\mathsf{Nat}] \mid \mathtt{suma}(\mathsf{asa}, \mathsf{asa}) \mid \mathtt{prod}(\mathsf{asa}, \mathsf{asa})$$

Pero debe quedar claro que se trata de la sintaxis abstracta y no concreta.

Obsérvese ahora que la función de evaluación se simplifica considerablemente.

$$\begin{aligned} eval(\mathtt{num}[n]) &= n \\ eval(\mathtt{suma}(e_1, e_2) &= eval(e_1) + eval(e_2) \\ eval(\mathtt{prod}(e_1, e_2)) &= eval(e_1) * eval(e_2) \end{aligned}$$

2. La relación entre las sintaxis concreta y abstracta

Veamos ahora como están relacionados ambos niveles de sintaxis.

2.1. Análisis sintáctico

El proceso de traducción de la sintaxis concreta a la sintaxis abstracta se conoce como *análisis sintáctico*[2]. Un analizador sintáctico debe verificar si un programa es correcto con respecto a la sintaxis concreta y representarlo como un árbol de sintaxis abstracta. Nosotros lo definiremos mediante un juicio que relacione ambos niveles de sintaxis.

$$e_1\ \mathsf{E} \longleftrightarrow e_2\ \mathsf{asa}$$

de tal forma que por ejemplo se cumpla:

$$1 + (2 * 3)\ \mathsf{E} \longleftrightarrow \mathtt{suma}\Big(\mathtt{num}[1], \mathtt{prod}\big(\mathtt{num}[2], \mathtt{num}[3]\big)\Big)\ \mathsf{asa}$$

La relación de análisis sintáctico se define en base a las reglas de la definición inductiva de la sintaxis concreta como sigue:

[2]La palabra en inglés es *parsing*.

$$\frac{}{0\ \mathsf{D} \longleftrightarrow 0\ \mathsf{Nat}} \cdots\cdots \frac{}{9\ \mathsf{D} \longleftrightarrow 9\ \mathsf{Nat}}$$

$$\frac{s\ \mathsf{D} \longleftrightarrow k\ \mathsf{Nat}}{s\ \mathsf{N} \longleftrightarrow k\ \mathsf{Nat}} \qquad \frac{s_1\ \mathsf{N} \longleftrightarrow k_1\ \mathsf{Nat} \quad s_2\ \mathsf{D} \longleftrightarrow k_2\ \mathsf{Nat}}{s_1 s_2\ \mathsf{N} \longleftrightarrow 10k_1 + k_2\ \mathsf{Nat}}$$

$$\frac{s\ \mathsf{T} \longleftrightarrow t\ \mathsf{asa}}{s\ \mathsf{E} \longleftrightarrow t\ \mathsf{asa}} \qquad \frac{s_1\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{asa} \quad s_2\ \mathsf{T} \longleftrightarrow t_2\ \mathsf{asa}}{s_1 + s_2\ \mathsf{E} \longleftrightarrow \mathtt{suma}(t_1, t_2)\ \mathsf{asa}}$$

$$\frac{s\ \mathsf{F} \longleftrightarrow t\ \mathsf{asa}}{s\ \mathsf{T} \longleftrightarrow t\ \mathsf{asa}} \qquad \frac{s_1\ \mathsf{T} \longleftrightarrow t_1\ \mathsf{asa} \quad s_2\ \mathsf{F} \longleftrightarrow t_2\ \mathsf{asa}}{s_1 * s_2\ \mathsf{T} \longleftrightarrow \mathtt{prod}(t_1, t_2)\ \mathsf{asa}}$$

$$\frac{s\ \mathsf{N} \longleftrightarrow k\ \mathsf{Nat}}{s\ \mathsf{F} \longleftrightarrow \mathtt{num}[k]\ \mathsf{asa}} \qquad \frac{s\ \mathsf{E} \longleftrightarrow t\ \mathsf{asa}}{(s)\ \mathsf{F} \longleftrightarrow t\ \mathsf{asa}}$$

Obsérvese que hay una regla de análisis sintáctico por cada regla de la gramática de la sintaxis concreta.

Una vez dada la especificación debemos verificar que se cumplen las propiedades básicas esperadas. En este caso debemos verificar que las cadenas y árboles asa involucrados en la traducción $\longleftrightarrow$ realmente pertenecen a estas categorias

Proposición 1 *Se cumplen las siguientes propiedades de correctud para el analizador sintáctico.*

- *Si* $s\ \mathsf{D} \longleftrightarrow k\ \mathsf{Nat}$ *entonces* $s\ \mathsf{D}\ y\ k\ \mathsf{Nat}$.
- *Si* $s\ \mathsf{N} \longleftrightarrow k\ \mathsf{Nat}$ *entonces* $s\ \mathsf{N}\ y\ k\ \mathsf{Nat}$.
- *Si* $s\ \mathsf{E} \longleftrightarrow t\ \mathsf{asa}$ *entonces* $s\ \mathsf{E}\ y\ t\ \mathsf{asa}$.
- *Si* $s\ \mathsf{T} \longleftrightarrow t\ \mathsf{asa}$ *entonces* $s\ \mathsf{T}\ y\ t\ \mathsf{asa}$.
- *Si* $s\ \mathsf{F} \longleftrightarrow t\ \mathsf{asa}$ *entonces* $s\ \mathsf{F}\ y\ t\ \mathsf{asa}$.

Demostración. Diversas inducciones que el lector debe verificar ⊣

Obsérvese que el juicio $\longleftrightarrow$ es bidireccional por lo que permite hacer y deshacer el análisis sintáctico.

Definición 1 *El análisis sintáctico*[3] *de una cadena* s *consisten en hallar una expresión* t *tal que se cumple* $s\ E \longleftrightarrow t\ \mathsf{asa}$. *De no existir tal* t *decimos que el analizador falla.*

Esta definición se extiende a las otras categorías sintácticas de las gramáticas. El análisis sintáctico de nuestro lenguaje de expresiones aritméticas se resume en la siguiente

Proposición 2 *Se cumplen las siguientes propiedades:*

1. *Si* $s\ \mathsf{D}$ *entonces existe un* k *tal que* $s\ \mathsf{D} \longleftrightarrow k\ \mathsf{Nat}$.

[3] parsing.

2. *Si* s N *entonces existe un* k *tal que* s $\mathsf{N} \longleftrightarrow k$ Nat.

3. *Si* s E *entonces existe un* t *tal que* s $\mathsf{E} \longleftrightarrow t$ asa.

4. *Si* s T *entonces existe un* t *tal que* s $\mathsf{T} \longleftrightarrow t$ asa.

5. *Si* s F *entonces existe un* t *tal que* s $\mathsf{F} \longleftrightarrow t$ asa.

Demostración. Diversas inducciones.

⊣

Es importante observar que un analizador sintáctico debe cumplir las siguientes propiedades:

- Debe ser total en todas las expresiones correctas de la sintaxis concreta. Es decir, para cada s E debe existir un t asa tal que s $\mathsf{E} \longleftrightarrow t$ asa.
- No debe ser ambiguo, es decir, no deben existir una expresión s E y dos árboles distintos t_1, t_2 tales que s $\mathsf{E} \longleftrightarrow t_1$ asa y s $\mathsf{E} \longleftrightarrow t_2$ asa

2.2. Inversión del análisis sintáctico

Dado que nuestro juicio para análisis sintáctico es bidireccional podemos definir la inversión del análisis sintáctico (*unparsing*).

Definición 2 *Sea* t *un árbol de sintaxis abstracta, es decir,* t asa. *La inversión del análisis sintáctico para* t *consiste en hallar una expresión* e *tal que* e $\mathsf{E} \longleftrightarrow t$ asa.

El proceso de inversión es importante para la llamada *impresión con formato*[4] proceso que consiste en convertir una expresión en sintaxis abstracta en una cadena legible que en la mayoría de los casos no corresponde al programa en sintaxis concreta original al agregar formatos como espacios, paréntesis o comillas. Por ejemplo dado el asa $\mathtt{prod}(\mathtt{num}[3], \mathtt{prod}(\mathtt{num}[4], \mathtt{num}[5]))$ el proceso de impresión con formato podría producir la cadena `‘‘3 * 4 * 5’’`.

Por lo general el proceso de inversión es total e inherentemente ambiguo, es decir, cualquier expresión en sintaxis abstracta puede invertirse y ser escrita de diversas maneras. Un ejemplo de la ambigüedad es la inserción de paréntesis redundantes. De modo que cualquier implementación de la inversión del análisis sintáctico debe usar heurísticas para elegir entre distintas alternativas de representación para cadenas.

3. Extensión de EA con variables

El lenguaje de expresiones aritméticas EA que hemos desarrollado hasta ahora nos ha servido para ilustrar aspectos del diseño de la sintaxis de un lenguaje de programación, por ejemplo la importancia de la distinción entre la sintaxis concreta y abstracta. Sin embargo, este lenguaje es aun demasiado simple para estudiar otros fenómenos o conceptos. Uno de ellos, quizas el concepto

[4]En inglés *pretty printing*, traducción libre mia.

fundamental más importante, es el de *variable*, en particular las nociones de variable ligada y alcance. Para estudiar este concepto de forma aislada extendemos nuestro lenguaje con una nueva expresión para nombrar resultados preliminares, las expresiones `let`.

3.1. Expresiones `let`

Las expresiones `let` son un mecanismo para introducir variables con alcance restringido, es decir, variables locales. Nos interesan en este curso pues son mucho más simples de implementar que por ejemplo los bloques de un lenguaje imperativo como C. Este mecanismo permite nombrar valores o subexpresiones de una expresión dada con una variable local, de manera que resulte una expresión más eficiente con respecto a la evaluación pues el valor ligado a la variable local sólo se calcula por lo general sólo se realiza una vez. Veamos la definición incluyendo una descripción informal de su semántica, de la manera usual en manuales de lenguajes de programación:

- Sintaxis: `let` $x = e_1$ `in` e_2 `end`
- Semántica: para evaluar una expresión `let` primero debemos evaluar e_1 cuyo resultado se liga a la variable x y finalmente evaluar e_2 usando el valor actualizado de x. Dicho valor de e_2 es también el valor de toda la expresión `let`.

Por ejemplo, la expresión `let` $x = 2 * 3$ `in` $x + x$ `end` se evalua a 12, pero resulta más simple de analizar que la expresión $(2 * 3) + (2 * 3)$ cuyo proceso de evaluación obliga a calcular el valor de la operación $2 * 3$ dos veces.

Para agregar esta clase de expresiones a nuestro lenguaje EA, modificamos la sintaxis concreta como sigue, agregando una nueva categoría para las variables y extendiendo la categoría de factores con expresiones `let`.

$$\begin{array}{lcl} E & ::= & \ldots \\ T & ::= & \ldots \\ F & ::= & V \mid N \mid (E) \mid \texttt{let}\ V = E\ \texttt{in}\ E\ \texttt{end} \\ N & ::= & \ldots \\ D & ::= & \ldots \\ V & ::= & < identificador > \end{array}$$

Los juicios correspondientes para la sintaxis concreta son

$$\frac{s \quad < identificador >}{s\ \mathsf{V}} \qquad \frac{s\ \mathsf{V}}{s\ \mathsf{F}} \qquad \frac{x\ \mathsf{V} \quad e_1\ \mathsf{E} \quad e_2\ \mathsf{E}}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \mathsf{F}}$$

Ignoramos aquí los detalles de la definición de identificador, pertenecientes mas bien a la sintaxis léxica del lenguaje. Debe quedar claro que tal definición debe excluir palabras reservadas, símbolos especiales, como los de operador, números, etc.

En una implementación real el analizador léxico se encarga de descomponer la cadena de entrada en palabras reservadas, símbolos especiales, números e identificadores que posteriormente son procesados por el analizador sintáctico.

Una primera idea para representar variables en la sintaxis abstracta consiste en agregar un operador de índice cero para variables y un nuevo operador ternario $let(x, e_1, e_2)$ para expresiones `let`, donde x es una variable y e_1, e_2 son expresiones aritméticas.

Con tal idea en mente proponemos los siguientes juicios:

$$\frac{t \quad < identificador >}{\texttt{var}[t]\ \texttt{asa}} \qquad \frac{\texttt{var}[x] \quad t_1\ \texttt{asa} \quad t_2\ \texttt{asa}}{\texttt{let}(\texttt{var}[x], t_1, t_2)\ \texttt{asa}}$$

3.2. Ligado y alcance de variables

La introducción de expresiones *let* en nuestro lenguaje introduce el concepto de ligado y alcance de una variable, estos conceptos se estudiaron con anterioridad en el curso de análisis lógico en el ámbito de la lógica de predicados.

Considérese la siguiente expresión

$$\texttt{let}\ x = 1\ \texttt{in}\ (\texttt{let}\ x = x + 1\ \texttt{in}\ x + x\ \texttt{end})\ \texttt{end}$$

Para concluir que el valor de esta expresión `let` anidada es 4, necesitamos distinguir cuales presencias de la variable x están ligadas y en tal caso dentro de que alcance se encuentran. Esto no queda claro con nuestra representación en la sintaxis abstracta. El árbol de sintaxis abstracta correspondiente es

$$\texttt{let}(\texttt{var}[x], \texttt{num}[1], \texttt{let}(\texttt{var}[x], \texttt{suma}(\texttt{var}[x], \texttt{num}[1]), \texttt{suma}(\texttt{var}[x], \texttt{var}[x])))$$

y no permite distinguir de manera clara el ligado y alcance, debido a que se está usando una misma variable con dos alcances y ligados distintos.

Cualesquiera que sean las reglas de ligado y alcance para expresiones, deben permitir que la expresión anterior sea equivalente a

$$\texttt{let}\ x_1 = 1\ \texttt{in}\ \texttt{let}\ x_2 = x_1 + 1\ \texttt{in}\ x_2 + x_2\ \texttt{end}\ \texttt{end}$$

la cual resulta más simple de entender y nos deja claramente que el alcance de la variable x en cualquier expresión de la forma $\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$ es la expresión e_2 y no la expresión e_1.

Veamos otro par de ejemplos:

- En la expresión $\texttt{let}\ x = y\ \texttt{in}\ \texttt{let}\ y = 2\ \texttt{in}\ x\ \texttt{end}\ \texttt{end}$ el alcance de "`let` x" es $\texttt{let}\ y = 2\ \texttt{in}\ x\ \texttt{end}$ mientras que el alcance de "`let` y" es la segunda presencia de x. Por otro lado la primera presencia de y está libre y fuera del alcance de "`let` y". De acuerdo a esto la expresión completa debe evaluarse a y.
- En la expresión $\texttt{let}\ x = 5\ \texttt{in}\ \texttt{let}\ x = 3\ \texttt{in}\ x + x\ \texttt{end}\ \texttt{end}$ el ligado exterior de x es vacuo pues todas las presencias de x son ligadas por el "`let` x" interno, por lo que la expresión se evalua al mismo valor que $\texttt{let}\ x = 3\ \texttt{in}\ x + x\ \texttt{end}$, es decir, al valor 6.

El proceso para determinar a qué alcance pertenece una presencia particular de una variable en una expresión `let` se conoce como *resolución del alcance*. Dependiendo de cómo se implemente, la resolución del alcance puede ser *estática* o *léxica* en cuyo caso una variable siempre se refiere a su

ambiente inmediato superior en el texto, lo cual puede verificarse en tiempo de compilación, o bien *dinámica*, asociando a cada variable con una pila global de ligados de manera que la evaluación siempre se refiere al ligado en el tope de dicha pila. Estos conceptos se estudiarán con más detalle más adelante.

De los ejemplos anteriores podemos concluir que hay dos procesos importantes que definir e implementar, el renombre de variables y la resolución del alcance. Desafortunadamente la representación actual de las expresiones `let` en la sintaxis abstracta es inadecuada para implementar ambos procesos. Por ejemplo no es claro como implementar el renombre de variables ligadas. Considérense las siguientes expresiones:

$$\texttt{let } x = 1 \texttt{ in let } y = 2 \texttt{ in } x + y \texttt{ end end}$$

$$\texttt{let } y = 1 \texttt{ in let } x = 2 \texttt{ in } y + x \texttt{ end end}$$

Ambas deben considerarse equivalentes pues sólo difieren en el orden en que figuran las variables ligadas, no en su valor que es 3.
Con nuestra propuesta actual, los árboles de sintaxis abstracta correspondientes son

$$\texttt{let}(\texttt{var}[x], \texttt{num}[1], \texttt{let}(\texttt{var}[y], \texttt{num}[2], \texttt{suma}(\texttt{var}[x], \texttt{var}[y])))$$

$$\texttt{let}(\texttt{var}[y], \texttt{num}[1], \texttt{let}(\texttt{var}[x], \texttt{num}[2], \texttt{suma}(\texttt{var}[y], \texttt{var}[x])))$$

que son estructuralmente distintos y no es claro como definir una relación de igualdad de manera que resulten equivalentes.

Los problemas de nuestra representación pueden resumirse en la imposibilidad de distinguir entre la definición y el uso de una presencia de variable. Estos obstaculos se resuelven al representar explicitamente el ligado de variables.

3.3. Sintaxis abstracta de orden superior

La técnica de sintaxis abstracta de orden superior permite codificar información acerca del alcance y ligado de una variable de manera uniforme. La idea básica consiste en modelar las variables de lenguaje con variables en el metalenguaje, agregando al lenguaje de la sintaxis abstracta un constructor de árboles $x.t$ que denota el ligado de la variable x en el árbol t. Cualquier presencia de x en t que no esté afectada por otro ligado $x.t'$ se considera ligada por $x.t$. Tal constructor se conoce como *abstracción*. Las variables de una abstracción son un nuevo concepto primitivo por agregar a la sintaxis abstracta. De esta manera la categoría de variables se distingue de otras categorías de árboles abstractos.

Una expresión de orden superior puede ser:

- Una variable: x, y, z,etc.
- Una aplicación de operador a otras expresiones $o(t_1, \ldots, t_n)$.

$$\texttt{num}[4], \texttt{suma}(x, \texttt{num}[1]),\ \textit{etc.}$$

- Una abstracción $x.t$ donde x es una variable que se considera ligada en la expresión t.

$$x.\,\texttt{suma}(x, \texttt{num}[1]),\ x.y.\,\texttt{suma}(x, y),\ etc$$

En esta representación cada expresión se conoce también como árbol de ligado abstracto (ala). En nuestro caso particular su definición es:

$$\frac{}{x\ \mathsf{ala}} \qquad \frac{n\ \mathsf{Nat}}{\texttt{num}[n]\ \mathsf{ala}}$$

$$\frac{t_1\ \mathsf{ala} \quad t_2\ \mathsf{ala}}{\texttt{suma}(t_1, t_2)\ \mathsf{ala}} \qquad \frac{t_1\ \mathsf{ala} \quad t_2\ \mathsf{ala}}{\texttt{prod}(t_1, t_2)\ \mathsf{ala}} \qquad \frac{t_1\ \mathsf{ala} \quad t_2\ \mathsf{ala}}{\texttt{let}(t_1, x.t_2)\ \mathsf{ala}}$$

Obsérvese que las variables se consideran como primitivas, no se verifica que x es una variable sino que está implicito que lo es. Mas aún no hay un proceso explícito de traducción de un identificador a variable (un operador `var`). Se supone que los identificadores se traducen en variables durante el proceso de análisis sintáctico.

Un árbol de ligado abstracto tiene como hojas a las variables y a los operadores donde $ar(o) = 0$, en este caso el operador `num`.

Enfatizamos aquí que las reglas para la resolución del alcance así como el ligado en expresiones `let` estan codificadas directamente en la sintaxis abstracta de orden superior.

Como ejemplo la expresión concreta

$$\texttt{let}\ x_1 = 1\ \texttt{in let}\ x_2 = x_1 + 1\ \texttt{in}\ x_2 + x_2\ \texttt{end end}$$

se representa con

$$\texttt{let}(\texttt{num}[1], x_1.\texttt{let}(\texttt{suma}(x_1, \texttt{num}[1]), x_2.\,\texttt{suma}(x_2, x_2)))$$

y el mecanismo de sintaxis de orden superior permite reconocer de inmediato los ligados y alcances.

Finalmente es posible definir el juicio de equivalencia entre la sintaxis concreta y la sintaxis abstracta de orden superior (ala) de manera análoga a la definición con asa. El caso de expresiones `let` es:

$$\frac{e_1\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{ala} \qquad e_2\ \mathsf{E} \longleftrightarrow t_2\ \mathsf{ala}}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \ \mathsf{F} \longleftrightarrow \texttt{let}(t_1, x.t_2)\ \mathsf{ala}}$$

En adelante usaremos siempre la sintaxis abstracta de orden superior por lo que cuando hablemos de un árbol de sintaxis abstracta asa entenderemos en caso necesario que se trata de un árbol de ligado abstracto.

4. Sustitución y α-equivalencia

Para finalizar nuestra discusión sobre la sintaxis vamos a definir formalmente dos conceptos fundamentales, la sustitución y la α-equivalencia. La imposibilidad de definir estos conceptos de forma adecuada con nuestra primera propuesta de sintaxis abstracta nos ha llevado a emplear la sintaxis abstracta de orden superior con la cual es muy simple definir la operación de sustitución. Esta operación será la herramienta fundamental para estudiar estrategias de evaluación así como los mecanismos de paso de parámetros.

Definición 3 *Decimos que dos expresiones e_1, e_2 son α-equivalentes y escribimos $e_1 \equiv_\alpha e_2$ si y sólo si e_1 y e_2 difieren únicamente en los nombres de variables ligadas*

Por ejemplo se cumple que

$$\texttt{let } x = 3 \texttt{ in } x + 1 \texttt{ end} \equiv_\alpha \texttt{let } y = 3 \texttt{ in } y + 1 \texttt{ end}$$

Por otra parte la operación de sustitución consiste en reemplazar las presencias libres de una variable x en una expresión e por otra expresión e'. Esta operación se debe definir con sumo cuidado cuidando que ninguna presencia libre de una varable se ligue después de aplicar la sustitución, en este sentido el mecanismo de sustitución es muy similar al empleado en la lógica de predicados de primer orden.

La operación de sustitución se denotará con $e[x := e']$. Otras notaciones comunes son $e[x \mapsto e']$ y $\{x/e'\}e$. Antes de dar la definición formal veamos unos ejemplos:

- $(5 * x + 7)[x := 2 * y]$, en este caso no hay ligado alguno por lo que la sustitución es textual

  $$(5 * x + 7)[x := 2 * y] = 5 * (2 * y) + 7$$

- $(\texttt{let } y = 5 \texttt{ in } y * x + 7 \texttt{ end})[x := 2 * y]$. En este caso hay que tener cuidado, la expresión a sustituirse contiene a la variable ligada y por lo que si devolvemos como resultado

  $$(\texttt{let } y = 5 \texttt{ in } y * x + 7 \texttt{ end})[x := 2 * y] = \texttt{let } y = 5 \texttt{ in } y * (2 * y) + 7 \texttt{ end} \quad (Incorrecto)$$

  estamos cometiendo un error ya que en la posición de la variable libre x tendriamos ahora una presencia ligada de y. Permitir este tipo de sustituciones causaría entre otras cosas que expresiones α-equivalentes dejaran de serlo al aplicar sustitución. Por ejemplo, se tiene

  $$(\texttt{let } z = 5 \texttt{ in } z * x + 7 \texttt{ end})[x := 2 * y] = \texttt{let } z = 5 \texttt{ in } z * (2 * y) + 7 \texttt{ end}$$

  pero los resultados ya no son equivalentes pues en el primer caso la expresión let se evalua a 57 mientras que en el segundo caso se simplifica a $10 * y + 7$ que no se puede evaluar pues no conocemos el valor de y.

Teniendo en cuenta estas observaciones definimos al conjunto de variables libres de una expresión y la operación de sustitución que evita la captura de variables libres como sigue:

Definición 4 *Dada una expresión t, definimos el conjunto de variables libres de t, denotado $FV(t)$ como sigue:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(o(t_1, \ldots, t_n)) &= FV(t_1) \cup \ldots \cup FV(t_n) \\ FV(x.t) &= FV(t) \setminus \{x\} \end{aligned}$$

Definición 5 *La operación de sustitución $e[x := r]$ se define recursivamente como sigue:*

$$\begin{array}{rcl} x[x := r] & = & r \\ z[x := r] & = & z \;\; si\, x \neq z \\ o(t_1, \ldots, t_n)[x := r] & = & o(t_1[x := r], \ldots, t_n[x := r]) \\ (z.e)[x := r] & = & z.e[x := r] \;\; si \;\; z \neq x \;\; y \; z \notin FV(r) \end{array}$$

De esta definición se siguen en particular las siguientes propiedades que el lector debe verificar

- Si $x \notin FV(e)$ entonces $e[x := e'] = e$
- $(x.e)[x := t] = x.e$
- $x.e \equiv_\alpha z.e[x := z]$

Obsérvese que nuestra definición evita los problemas de captura de variables libres pero al precio de volverse una operación parcial. Por ejemplo $(y.x + y)[x := y * z]$ queda indefinida pues $y \in FV(y * z)$. Este problema se soluciona al emplear la α equivalencia conviniendo dos expresiones α-equivalentes se consideran idénticas, es decir, usando clases de expresiones α-equivalentes en vez de expresiones, situación que nunca hacemos explícita. De esta manera el problema anterior se resuelve como sigue:

$$(y.x + y)[x := y * z] \equiv_\alpha (w.x + w)[x := y * z] = w.(y * z) + w$$

De las definiciones generales recien discutidas se sigue la definición particular para el caso de las expresiones let, enunciado a continuación:

$$FV(\texttt{let}(t_1, x.t_2)) = FV(t_1) \cup (FV(t_2) \setminus \{x\})$$

$$\big(\texttt{let}(t_1, z.t_2)\big)[x := r] = \texttt{let}(t_1[x := r], z.t_2[x := r]) \;\; \text{si } x \neq z \text{ y } z \notin FV(r)$$

Hasta aquí hemos desarrollado los principales conceptos concernientes a la sintaxis de nuestro lenguaje de expresiones aritméticas.
Nuestro siguiente tema es la semántica de lenguajes de programación, en particular debemos definir una manera formal de evaluar un programa y a partir de esta definición implementar un intérprete para el lenguaje con variables.

5. Ejercicio

Considérese la siguiente descripción de la sintaxis de un lenguaje de expresiones numéricas elementales, llamado EL.

Un programa en el lenguaje EL es una terna cuya primera componente es la palabra reservada EL, seguida de un numeral n que especifica el número de argumentos (parámetros) del programa y de un cuerpo e que es una expresión numérica. Donde una expresión numérica es una de las siguientes:

- Una literal entera.
- Una operación de referencia a alguno de los parámetros de entrada, especificado por un numeral.[5]
- Una operación aritmética formada por la aplicación de un operador aritmético a dos expresiones numéricas. donde un operador aritmético es uno de los siguientes:
  - suma
  - resta
  - producto
  - división
  - residuo
- Un condicional que permite elegir entre dos expresiones numéricas de acuerdo a una expresión booleana que sirve de guardia. Donde una expresión booleana es una de las siguientes:
  - Una literal booleana.
  - Una operación relacional formada por la aplicación de un operador binario relacional a dos expresiones numéricas. Donde un operador binario relacional es uno de los siguientes:
    - Menor que
    - Igual
    - Mayor que
  - Una operación lógica formada por la aplicación de un operador lógico binario a dos expresiones booleanas. Donde un operador lógico binario es uno de los siguientes:
    - Conjunción
    - Disyunción

De acuerdo a esta definición realice lo siguiente:

1. Defina una sintaxis concreta para el lenguaje EL.
2. Defina la sintaxis abstracta para EL, correspondiente a la sintaxis concreta recién definida.
3. Defina el juicio de análisis sintáctico para EL.
4. Escriba los siguientes programas en EL, en ambas sintaxis:

   a) Un programa que reciba dos argumentos y devuelva el residuo del doble del primero al dividir entre, el segundo argumento menos 3.

   b) Un programa que reciba tres argumentos y en caso de que el primero sea un número mayor que 1 y menor que 10 devuelva cero, sino debe devolver el producto del segundo y tercer argumentos.

[5] Similar a la operación de acceso a una posición particular en un arreglo.