Lenguajes de Programación 2023-1
Nota de clase 3: Sintaxis
Introducción

Javier Enríquez Mendoza

22 de agosto de 2022

Como se vio en notas anteriores, la sintaxis de un lenguaje define las cadenas de texto o expresiones que pertenecen a un lenguaje. Con anterioridad se estudiaron herramientas que nos ayudan a definir formalmente la sintaxis de un lenguaje, como lo pueden ser gramáticas libres del contexto o incluso expresiones regulares. En esta nota se estudia formalmente el concepto de sintaxis para el caso especifico de los lenguajes de programación con el fin de entender la forma de construir expresiones pertenecientes a estos.

1 Sintaxis en Lenguajes de Programación

La sintaxis en los lenguajes de programación se refiere a la forma correcta de escribir las expresiones pertenecientes a este de tal forma que la computadora sea capaz de interpretar lo que dicha instrucción quiere decir.

Para definir la sintaxis se usan dos tipos de objetos:

- **Cadenas de texto**, que son las expresiones del lenguaje tal y como las programadoras y programadores las escriben en su código.
- **Árboles de sintaxis abstracta**, se trata de una representación como estructuras arbóreas de las expresiones de lenguaje en donde se describe su estructura jerárquica.

Dependiendo del objeto utilizado se definen dos niveles distintos para el estudio de la sintaxis de los lenguajes de programación: la **sintaxis concreta** y la **sintaxis abstracta**.

1.1 Sintaxis Concreta

En la sintaxis concreta el objeto de estudio son las cadenas de texto que pertenecen al lenguaje. Esta sintaxis es diseñada con las usuarias y usuarios finales del lenguaje en mente, por lo que debe estar bien estructurada y ser fácil de leer y escribir.

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

- Sintaxis léxica
- Sintaxis libre de contexto

Definición 1.1 (Sintaxis léxica)**.** Es la parte de la sintaxis concreta que describe la construcción de lexemas (átomos, tokens, símbolos terminales).

Los principales lexemas en un lenguaje de programación son: palabras reservadas, identificadores, números, literales, espacios, etc.

La principal herramienta usada para la descripción de la sintaxis léxica son las expresiones regulares, mediante las cuales se capturan los patrones de escritura de los lexemas.

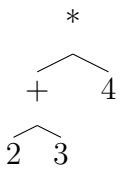
Definición 1.2 (Sintaxis libre de contexto)**.** Es la parte de la sintaxis concreta que describe la construcción de frases del lenguaje.

Las frases descritas por la sintaxis libre del contexto constituyen los programas escritos en el lenguaje de programación. La herramienta principal para la definición son las gramáticas libres del contexto, generalmente en forma de Backus-Naur (BNF).

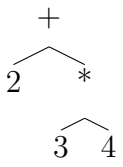
Ejemplo 1.1 (Sintaxis concreta para el lenguaje de expresiones aritméticas)**.** En este ejemplo se define formalmente la sintaxis concreta del lenguaje EA descrito en la primera nota de clase, en donde se propuso la siguiente gramática libre del contexto para describir su sintaxis:

$$
\begin{array}{lcl}
e & ::= & n \mid e + e \mid e * e \\
n & ::= & d \mid nd \\
d & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Sin embargo se puede observar que la gramática anterior es ambigua, esto quiere decir que para la misma cadena se pueden dar dos derivaciones diferentes, lo cual cambia el significado de ésta. Por ejemplo la cadena $2 + 3 * 4$ se puede derivar de estas dos formas:



Que representa la expresión $(2 + 3) * 4$ que al evaluar nos da como resultado 20.



Que a su vez está representando la expresión $2 + (3 * 4)$ cuyo resultado es 14.
En general cuando se utilice una gramática libre de contexto para definir la sintaxis concreta de un lenguaje de programación se busca que no tenga ambigüedad. El eliminar la ambigüedad de una gramática no es un proceso algorítmico y no siempre es posible hacerlo. Sin embargo para

el lenguaje EA se puede eliminar la ambigüedad modelando la precedencia y asociatividad de los operadores.
Se jerarquiza la gramática en números, factores, términos y expresiones, obteniendo la siguiente gramática:

$$
\begin{array}{lcl}
e & ::= & t \mid e + t \\
t & ::= & f \mid t * f \\
f & ::= & n \mid (e) \\
n & ::= & d \mid nd \\
d & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

La gramática anterior define la sintaxis concreta para el lenguaje de expresiones aritméticas.

La sintaxis concreta puede representarse también usando definiciones inductivas, simplemente traduciendo la gramática del lenguaje a reglas de inferencia como se ve en el ejemplo siguiente.

Ejemplo 1.2 (Sintaxis concreta de EA mediante definiciones inductivas)**.** En este ejemplo se traduce la gramática dada en el ejemplo 1.1 a reglas de inferencia.

$$
\frac{}{0\ \mathsf{D}} \qquad \frac{}{1\ \mathsf{D}} \qquad \cdots \qquad \frac{}{9\ \mathsf{D}}
$$

$$
\frac{d\ \mathsf{D}}{d\ \mathsf{N}} \qquad \frac{n\ \mathsf{N} \quad d\ \mathsf{D}}{nd\ \mathsf{N}}
$$

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

$$
\frac{f\ \mathsf{F}}{F\ \mathsf{T}} \qquad \frac{t\ \mathsf{T} \quad f\ \mathsf{F}}{t * f\ \mathsf{T}}
$$

$$
\frac{t\ \mathsf{T}}{t\ \mathsf{E}} \qquad \frac{e\ \mathsf{E} \quad t\ \mathsf{T}}{e + t \quad \mathsf{E}}
$$

Se puede observar que cada regla de inferencia de la definición inductiva anterior corresponde con una regla de producción de la gramática dada en el ejemplo 1.1.

La sintaxis concreta define la primera representación de los programas escritos en el lenguaje de programación, la cual es útil para el desarrollo de programas pero no lo es para el razonamiento sobre estos. Es por esto que es traducida a una representación interna definida por la sintaxis abstracta.

1.2 Sintaxis Abstracta

La sintaxis abstracta provee una representación mas simple de las expresiones del lenguaje, esta sintaxis es mas sencilla de manipular y facilita la tarea de definir funciones sobre las expresiones del lenguaje, como una función de evaluación, así como el razonamiento inductivo sobre las expresiones ya que las definiciones inductivas para la sintaxis concreta suele ser muy complejas. Las expresiones son representadas mediante un árbol de sintaxis abstracta (asa).

Definición 1.3 (Árbol de sintaxis abstracta)**.** Es un árbol ordenado en donde los nodos internos se etiquetan por un operador principal de la expresión y sus hijos son los argumentos de la operación, es decir, el número de hijos de un nodo corresponde con la aridad del operador con el que está etiquetado.

El árbol de sintaxis abstracta captura el orden de ejecución de las operaciones de una expresión, ya que adopta una notación prefija en la que el primer operador de la expresión corresponde con la operación principal. Por lo que se elimina el uso de paréntesis que es necesario en la sintaxis concreta para eliminar la ambigüedad.

Observación. El árbol de sintaxis abstracta de una expresión es **único**, sin importar la representación que tenga en sintaxis concreta.

La sintaxis abstracta se puede definir con el uso de reglas de inferencia para juicios a asa que se lee como *a es un árbol de sintaxis abstracta.*

Ejemplo 1.3 (Definición de sintaxis abstracta para EA)**.** Para el caso de las expresiones aritméticas existen tres clases de expresiones: los números, sumas y productos, las cuales corresponden a tres operadores: `num`, `suma` y `prod` respectivamente. La aridad de `num` es 0 mientras que la de los operadores `suma` y `prod` es 2 pues se trata de operaciones binarias.

Se define la sintaxis abstracta de EA mediante las siguientes reglas:

$$\frac{n \in \mathbb{N}}{\texttt{num}[n]\ \text{asa}} \qquad \frac{t_1\ \text{asa} \quad t_2\ \text{asa}}{\texttt{suma}(t_1, t_2)\ \text{asa}} \qquad \frac{t_1\ \text{asa} \quad t_2\ \text{asa}}{\texttt{prod}(t_1, t_2)\ \text{asa}}$$

2 De la sintaxis concreta a la abstracta

La sintaxis concreta provee de una forma intuitiva y sencilla para que las desarrolladoras y desarrolladores del lenguaje puedan escribir programas en él. Mientras que la sintaxis abstracta permite el razonamiento sobre estos programas. Ambas cumplen con un propósito especifico y ambas son necesarias en un lenguaje de programación, pues sería conflictivo pedirle a las personas que programan en el lenguaje que lo hicieran en sintaxis abstracta, así como el tratar de razonar inductivamente con la sintaxis concreta es una tarea difícil.

Como primera tarea para el proceso de ejecución de un programa se debe definir una etapa de traducción entre la sintaxis concreta y la abstracta. La traducción entre estos dos niveles de sintaxis debe preservar el significado del programa. A esta etapa se le conoce como análisis sintáctico, también se emplea el término en ingles *parsing.*

El proceso de traducción de una expresión en sintaxis concreta a abstracta se divide en dos etapas:

- Análisis léxico
- Análisis sintáctico

Definición 2.1 (Análisis Léxico)**.** En el análisis léxico se procesa la expresión en sintaxis concreta carácter por carácter y se divide en una serie de unidades atómicas llamadas lexemas o *tokens* los cuales son clasificados en categorías. Se obtiene como resultado una representación intermedia de la cadena de entrada como una lista de lexemas.

Ejemplo 2.1 (Análisis Léxico)**.** Para la expresión $3 + 8 * 5$ en sintaxis concreta del lenguaje de expresiones aritméticas se construye la siguiente lista de lexemas:

$$[\texttt{Num}\ 3, \texttt{Oper}\ +, \texttt{Num}\ 8, \texttt{Oper}\ *, \texttt{Num}\ 5]$$

En donde se pueden distinguir dos categorías de lexemas: números y operadores.

Observación. El proceso de análisis léxico es importante en la implementación de un compilador o interprete sin embargo esta representación intermedia no aporta nada a la formalización de los programas por lo que se obviará esta etapa en la parte teórica del curso y se combinará con el análisis sintáctico.

Definición 2.2 (Análisis Sintáctico)**.** El análisis sintáctico es el proceso mediante el cuál a partir de una expresión e en sintaxis concreta se encuentra un árbol de sintaxis abstracta a tal que el juicio $e\ \mathsf{E} \longleftrightarrow a\ \mathsf{asa}$ se cumple. El árbol de sintaxis abstracta no siempre existe y de no encontrarse se dice que el analizador sintáctico fallo.

El analizador sintáctico debe verificar si un programa es correcto respecto a la sintaxis concreta del lenguaje en el que fue desarrollado y posteriormente devolver una representación del programa como un árbol de sintaxis abstracta.

Como se discutió con anterioridad las reglas de inferencia y los juicios no sólo se emplean para la definición de propiedades y conjuntos sino también se usan para definir relaciones entre objetos, por lo que usaremos esta herramienta para definir un analizador sintáctico mediante la relación:

$$e\ \mathsf{E} \longleftrightarrow a\ \mathsf{asa}$$

que es válida si y sólo si la expresión e en sintaxis concreta corresponde al árbol de sintaxis abstracta a.

Ejemplo 2.2 (Analizador Sintáctico para EA)**.** Se define el analizador sintáctico para el lenguaje de expresiones aritméticas, a partir de las reglas de sintaxis concreta del lenguaje, agregando la relación correspondiente en cada caso:

$$\frac{e\ \mathsf{N} \qquad e \in \mathbb{N}}{e\ \mathsf{F} \longleftrightarrow \texttt{num}[e]\ \mathsf{asa}} \qquad \frac{e\ \mathsf{E} \longleftrightarrow a\ \mathsf{asa}}{(e)\ \mathsf{F} \longleftrightarrow a\ \mathsf{asa}}$$

$$\frac{e\ \mathsf{F} \longleftrightarrow a\ \mathsf{asa}}{e\ \mathsf{T} \longleftrightarrow a\ \mathsf{asa}} \qquad \frac{e_1\ \mathsf{T} \longleftrightarrow a_1\ \mathsf{asa} \qquad e_2\ \mathsf{F} \longleftrightarrow a_2\ \mathsf{asa}}{e_1 * e_2\ \mathsf{T} \longleftrightarrow \texttt{prod}(a_1, a_2)\ \mathsf{asa}}$$

$$\frac{e\ \mathsf{T} \longleftrightarrow a\ \mathsf{asa}}{e\ \mathsf{E} \longleftrightarrow a\ \mathsf{asa}} \qquad \frac{e_1\ \mathsf{E} \longleftrightarrow a_1\ \mathsf{asa} \qquad e_2\ \mathsf{T} \longleftrightarrow a_2\ \mathsf{asa}}{e_1 + e_2\ \mathsf{T} \longleftrightarrow \texttt{suma}(a_1, a_2)\ \mathsf{asa}}$$

Observación. Estas reglas pueden interpretarse como la relación que existe entre la sintaxis concreta y la abstracta, pero también se pueden interpretar como un algoritmo de traducción entre los dos niveles de sintaxis.

En general un analizador sintáctico debe cumplir las siguientes proposiciones:

Proposición 2.3 (Total)**.** Un analizador sintáctico debe ser una función total sobre el dominio de las expresiones del lenguaje, es decir, para cada e E existe un a asa tal que se satisface el juicio e E $\longleftrightarrow$ a asa

Proposición 2.4 (Libre de Ambigüedad)**.** Un analizador sintáctico no debe ser ambiguo, esto es que si e E $\longleftrightarrow$ a_1 asa y e E $\longleftrightarrow$ a_2 asa entonces $a_1 = a_2$. En otras palabras, el árbol de sintaxis abstracta de una expresión es único.

3 El operador `let`

Ahora extendamos nuestro simple lenguaje de expresiones aritméticas con variables y el operador `let` para la definición de variables locales.

```
let x = 5 in x + 1 end
```

Para agregar un nuevo constructor de expresiones a nuestro lenguaje se debe extender el conjunto de reglas de sintaxis concreta y abstracta agregando las reglas correspondientes a el nuevo constructor.

Sintaxis Concreta

$$\frac{e \text{ identificador}}{e \text{ V}} \qquad \frac{e \text{ V}}{e \text{ F}} \qquad \frac{x \text{ V} \quad e_1 \text{ E} \quad e_2 \text{ E}}{\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end E}}$$

En donde se agregan los juicios v V para indicar que v es una variable y s identificador que dice que la cadena s es un identificador válido. Para definir a un identificador como válido se deben excluir palabras reservadas, operadores y constantes del lenguaje.

Sintaxis Abstracta Una primera idea para definir la sintaxis abstracta del operador es la siguiente

$$\frac{x \text{ identificador}}{\texttt{var}[x] \text{ asa}} \qquad \frac{\texttt{var}[x] \text{ asa} \quad a_1 \text{ asa} \quad a_2 \text{ asa}}{\texttt{let}(\texttt{var}[x], a_1, a_2) \text{ asa}}$$

Sin embargo estas reglas no están definiendo fielmente el comportamiento del operador `let`. Esto se debe a que la variable x debería estar relacionada con la expresión a_2 en el sentido de que las apariciones de x en a_2 alcanzan su valor con la definición de la expresión `let` lo cuál indica un ligado de x en a_2.

Más adelante en esta sección se da una definición de sintaxis abstracta mas adecuada para el operador `let`, pero primero es necesario definir algunos conceptos.

3.1 Ligado y Alcance de una variable

Al introducir el constructor `let` al lenguaje se introducen con él el concepto de ligado y alcance de una variable.

Definición 3.1 (Alcance)**.** El alcance de una variable es la región de un programa en la cual esta alcanza su valor.

En una expresión `let`

```
let x = v in ··· end
```

el alcance de la variable `x` se delimita por las palabras reservadas **`in`** y **`end`**, es decir, el alcance de `x` es toda la región del programa que se encuentra entre estas palabras reservadas.

No todas las apariciones de una variable en un programa definen el mismo comportamiento. Por ejemplo en la siguiente expresión:

```
let x = 5 in x + y end
```

se usan variables en tres ocasiones pero cada una representa un tipo de variable distinta.

Definición 3.2 (Variable de ligado)**.** La instancia de ligado de una variable es aquella que da a ésta su valor. Es decir, es la aparición de la variable en donde ésta es definida.

En la expresión `let` anterior la variable de ligado es la siguiente:

```
let x = 5 in x + y  end
```

Definición 3.3 (Variable ligada)**.** Una variable está ligada si se encuentra contenida dentro del alcance de una variable de ligado con su nombre.

En la expresión `let` anterior la variable ligada es:

```
let x = 5 in x + y end
```

pues se encuentra dentro del alcance de la variable de ligado `x`.

Definición 3.4 (Variable libre)**.** Una variable está libre en una expresión, si no se encuentra dentro del alcance de una variable de ligado con su nombre. Es decir, una variable es libre si no está ligada.

En la expresión `let` la variable libre es:

```
let x = 5 in x + y end
```

pues la variable `y` nunca se liga.

Para que la sintaxis abstracta del operador `let` sea fiel a la definición del operador es necesario que modele también el ligado de la variable en el cuerpo de expresión.

3.2 Sintaxis abstracta de orden superior

En la sintaxis abstracta utilizada hasta ahora, las variables son tratadas de la misma forma que los números siendo representadas por términos, esta sintaxis tiene el nombre de **sintaxis abstracta de primer orden**. Sin embargo las variables juegan un papel especial en nuestro lenguaje.

La sintaxis **abstracta de orden superior** define un tratamiento especial para las variables en donde éstas así como su alcance y ligado forman parte del meta-lenguaje con el que se define el árbol de sintaxis abstracta. Para esto se agrega el constructor $x.t$ que indica que la variable x está ligada en el árbol t, a este constructor se le llama *abstracción*.

Con este nuevo constructor se pueden definir correctamente las reglas de sintaxis abstracta del operador `let`, de tal forma que incluyan el ligado de la variable definida, con la siguiente regla:

$$\frac{a_1 \text{ asa} \qquad a_2 \text{ asa}}{\texttt{let}(a_1, x.a_2) \text{ asa}}$$

Hay que observar que las variables se consideran como primitivas, no se verifica que x es una variable sino que está implícito que lo es.

Ejemplos (Expresiones `let` en sintaxis abstracta)**.** Para cada una de las siguientes expresiones se construye su árbol de sintaxis abstracta de orden superior:

1. `let x = 5 in x * 2 end`

$$\texttt{let}(\texttt{num}[5], x.\texttt{suma}(x, \texttt{num}[2]))$$

2. `let x = 12 in let y = 2 in x * y end end`

$$\texttt{let}(\texttt{num}[12], x.\texttt{let}(\texttt{num}[2], y.\texttt{prod}(x, y)))$$

3. `let x = let y = 2 in y * y end in x + 5 end`

$$\texttt{let}(\texttt{let}(\texttt{num}[2], y.\texttt{prod}(y, y)), x.\texttt{suma}(x, 5))$$

4 Sustitución y α-equivalencias

Consideremos las siguientes expresiones:

$$\texttt{let x} = 3 \texttt{ in x} + 2 \texttt{ end} \qquad \texttt{let y} = 3 \texttt{ in y} + 2 \texttt{ end}$$

Ambas expresiones representan exactamente el mismo cómputo, sin embargo no se trata de la misma expresión pues sintacticamente son distintas. La diferencia entre las expresiones es únicamente en la elección del identificador para la variable definida dentro del `let`, es decir, podemos hacer que ambas expresiones sean iguales simplemente cambiando el nombre de la variable. A esto es a lo que se le conoce como α-equivalencia.

Definición 4.1 (α-equivalencia)**.** Se dice que dos expresiones e_1 y e_2 son α-equivalentes si y sólo si difieren únicamente en el nombre de las variables ligadas. Y se escribe $e_1 \equiv_\alpha e_2$.

Como su nombre lo sugiere la relación de α-equivalencia es una relación de equivalencia, por lo que es:

- **Reflexiva**: para toda expresión e, $e \equiv_\alpha e$.
- **Transitiva**: para todas las expresiones e_1, e_2 y e_3, si $e_1 \equiv_\alpha e_2$ y $e_2 \equiv_\alpha e_3$ entonces $e_1 \equiv_\alpha e_3$.
- **Simétrica**: para todas expresiones e_1 y e_2, si $e_1 \equiv_\alpha e_2$ entonces $e_2 \equiv_\alpha e_1$.

Si queremos conocer el valor de una expresión `let`, una primera idea para evaluar la expresión se sirve de la sustitución textual para simplificar la expresión sustituyendo el valor de las variables en sus ocurrencias ligadas en el cuerpo de la expresión. Primero se define el conjunto de variables libres de una expresión.

Definición 4.2 (Variables libres de una expresión)**.** Dada una expresión en sintaxis abstracta a definimos el conjunto de variables libres de la expresión, denotado $FV(a)$, como sigue:

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(O(a_1, \cdots, a_n)) & = & FV(a_1) \cup \cdots \cup FV(a_n) \\ FV(x.a) & = & FV(a) \backslash \{x\} \end{array}$$

Utilizando esta definición se define la función de sustitución para las expresiones de nuestro lenguaje.

Definición 4.3 (Función de Sustitución)**.** Se define la función de sustitución sobre los árboles de sintaxis abstracta de orden superior como sigue:

$$\begin{array}{lcll} x[x := e] & = & e & \\ z[x := e] & = & z & \\ \texttt{num}[n][x := e] & = & n & \\ \texttt{suma}(a_1, a_2)[x := e] & = & \texttt{suma}(a_1[x := e], a_2[x := e]) & \\ \texttt{prod}(a_1, a_2)[x := e] & = & \texttt{prod}(a_1[x := e], a_2[x := e]) & \\ \texttt{let}(a_1, a_2)[x := e] & = & \texttt{let}(a_1[x := e], a_2[x := e]) & \\ (z.a)[x := e] & = & z.(a[x := e]) & \text{Si } x \neq z \text{ y } z \notin FV(e) \\ (z.a)[x := e] & = & \texttt{indefinido} & \text{Si } z \in FV(e) \end{array}$$

Observemos que se puede generalizar el caso de los operadores en el siguiente

$$O(a_1, \cdots, a_n)[x := e] \quad = \quad O(a_1[x := e], \cdots, a_n[x := e])$$

En donde O es un operador de aridad n

A partir de α-equivalencias se puede renombrar las variables ligadas de una expresión y de esa forma evitar el error `indefinido` para el caso de las expresiones $z.a$.

Ejemplo 4.1 (Sustitución)**.** Se resuelve la siguiente sustitución.

$$
\begin{aligned}
& \texttt{let}(\texttt{suma}(\texttt{num}[2], x), y.\texttt{prod}(y, x))[x = \texttt{prod}(y, \texttt{num}[3])] \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], x)[x = \texttt{prod}(y, \texttt{num}[3])], y.\texttt{prod}(y, x)[x = \texttt{prod}(y, \texttt{num}[3])]) \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2][x = \texttt{prod}(y, \texttt{num}[3])], x[x = \texttt{prod}(y, \texttt{num}[3])]), y.\texttt{prod}(y, x)[x = \texttt{prod}(y, \texttt{num}[3])]) \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], x[x = \texttt{prod}(y, \texttt{num}[3])]), y.\texttt{prod}(y, x)[x = \texttt{prod}(y, \texttt{num}[3])]) \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], \texttt{prod}(y, \texttt{num}[3])), y.\texttt{prod}(y, x)[x = \texttt{prod}(y, \texttt{num}[3])]) \\
\equiv_\alpha \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], \texttt{prod}(y, \texttt{num}[3])), w.\texttt{prod}(w, x)[x = \texttt{prod}(y, \texttt{num}[3])]) \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], \texttt{prod}(y, \texttt{num}[3])), w.\texttt{prod}(w[x = \texttt{prod}(y, \texttt{num}[3])], x[x = \texttt{prod}(y, \texttt{num}[3])])) \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], \texttt{prod}(y, \texttt{num}[3])), w.\texttt{prod}(w, x[x = \texttt{prod}(y, \texttt{num}[3])])) \\
= \; & \texttt{let}(\texttt{suma}(\texttt{num}[2], \texttt{prod}(y, \texttt{num}[3])), w.\texttt{prod}(w, \texttt{prod}(y, \texttt{num}[3])))
\end{aligned}
$$

Puede utilizarse sintaxis concreta para simplificar la notación, dejando claro que eso representa un abuso de notación ya que el algoritmo está definido sobre los árboles de sintaxis abstracta.

Utilizando la función de sustitución se puede dar una primera idea de la evaluación de expresiones del lenguaje, sin embargo por si sola la sustitución textual no es suficiente para estudiar la semántica de nuestro lenguaje. El estudio de la semántica de un lenguaje requiere del uso de otros formalismos los cuales se definirán en la siguiente nota de clase.

Referencias

[1] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.

[2] Reyes Cabello L., Un lenguaje para expresiones aritméticas (EA) Construcción de un interprete y un Compilador, Universidad Autónoma de la Ciudad de México, 2020.

[3] Ramírez Pulido K., Soto Romero M., Enríquez Mendoza J., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-2

[4] Keller G., O'Connor-Davis L., Class Notes from the course Concepts of programming language design, Department of Information and Computing Sciences, Utrecht University, The Netherlands, Fall 2020.

[5] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.

[6] Mitchell J., Foundations for Programming Languages. MIT Press 1996.

[7] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.