Lenguajes del Programación
Unidad 2: Especificación Formal de los Lenguajes de Programación

Sintaxis Abstracta

Manuel Soto Romero*

Semestre 2026-1
Facultad de Ciencias UNAM

En la nota de clase anterior revisamos la formalización de la sintaxis concreta de un lenguaje de programación. Este nivel de especificación resulta indispensable para establecer cómo debe escribirse el código fuente; sin embargo, no basta para expresar toda la lógica y la estructura asociadas a un programa. La sintaxis concreta incorpora numerosos detalles (como delimitadores, espacios en blanco o paréntesis) que son esenciales para la interpretación del código, pero que no contribuyen de manera directa a la comprensión de su lógica fundamental. De hecho, estos elementos pueden dificultar el análisis y la transformación del código, especialmente cuando se requieren optimizaciones durante el proceso de traducción.

En esta nota abordaremos un segundo nivel de especificación: la sintaxis abstracta. A diferencia de la anterior, esta ofrece una representación simplificada y estructural del código fuente, enfocada en capturar la lógica y la jerarquía del programa sin incluir los detalles superficiales propios de la sintaxis concreta. La forma más común de representarla es mediante un Árbol de Sintaxis Abstracta (ASA), el cual facilita tanto el análisis como la manipulación del código en las fases de compilación o interpretación.

La formalización de la sintaxis abstracta es clave en el diseño de lenguajes de programación, ya que permite desarrollar herramientas de análisis y compilación más eficientes y robustas. Una representación clara y simplificada del programa favorece la detección de errores, la optimización del código y la incorporación de nuevas funcionalidades al lenguaje. Además, al proporcionar una base estándar para el análisis sintáctico, contribuye a la interoperabilidad entre distintas herramientas y entornos de desarrollo, fortaleciendo la cohesión[1] y eficiencia del ecosistema de programación.

1. Un ejemplo para ganar intuición

Supongamos que tenemos una calculadora simple que soporta operaciones aritméticas básicas: suma (`+`) y multiplicación (`*`), y utilizamos paréntesis para definir la precedencia. Tomemos el siguiente programa como ejemplo:

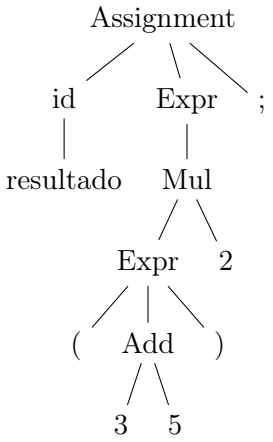
```
resultado = (3 + 5) * 2;
```

*Un agradecimiento a Juan Mario Sosa Romo por sus aportes durante la revisión de esta nota como parte de su servicio social.

[1] En el desarrollo de software, la cohesión se refiere al grado en que los elementos de un módulo o componente están relacionados y trabajan de manera conjunta para realizar una única tarea o propósito específico.

Árbol de Sintaxis Concreta

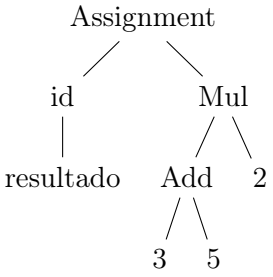
Podemos representar este programa en un árbol al que llamamos Árbol de Sintaxis Concreta (ASC). Este árbol incluye todos los detalles del código fuente como paréntesis y operadores:



En el ASC, cada elemento del código, incluyendo los paréntesis, se representa explícitamente, lo que hace que el árbol sea más detallado y complejo.

Árbol de Sintaxis Abstracta

El árbol de sintaxis abstracta omite los detalles innecesarios y se centra en la estructura lógica de la expresión:



En el ASA, los paréntesis no se representan explícitamente; en cambio, se conserva la estructura jerárquica de las operaciones, con la multiplicación en el nivel superior y la suma como una de sus operandos.

Beneficios de la sintaxis abstracta

Simplicidad y claridad El ASA es más simple y claro que el ASC, ya que omite los detalles sintácticos como los paréntesis, enfocándose en la estructura lógica de las operaciones.

Facilita el análisis y transformación La simplificación de la estructura facilita la implementación de análisis y optimizaciones. Por ejemplo, el compilador puede fácilmente identificar subexpresiones y aplicar optimizaciones algebraicas.

Reducción de complejidad Al eliminar elementos sintácticos redundantes, el ASA reduce la complejidad del procesamiento del código, mejorando la eficiencia de las herramientas de análisis.

2. Definición formal

A partir del ejemplo anterior, podemos definir formalmente a la sintaxis abstracta en términos de árboles, recordando de un árbol es un conjunto de nodos conectados por aristas y, en este caso, cada nodo representa una construcción del lenguaje de programación.

Definición 1 (*Árbol de Sintaxis Abstracta*)

Un ***árbol de sintaxis abstracta*** *es un árbol ordenado* $A = (N, A, R)$ *donde:*

- N *es un conjunto finito de nodos que representan las construcciones del lenguaje mediante etiquetas y las hojas representan a sus respectivos valores.*
- $A \subseteq N \times N$ *es un conjunto de aristas dirigidas que conectan a los nodos.*
- $R \in N$ *es la raíz del árbol.*

Representación mediante reglas de inferencia

Es posible representan a los ASA formalmente mediante reglas de inferencia que definen cómo se construyen los nodos del árbol a partir de sus subexpresiones. Las reglas de inferencia son una manera formal de especificar cómo las construcciones sintácticas de un lenguaje de programación se combinan para formar estructuras más complejas. Además nos permite especificar las reglas de construcción de los árboles de manera compacta.

Ejemplo 1: Sintaxis Abstracta de MiniLisp

Retomando el diseño de nuestro lenguaje MiniLisp, definiremos su sintaxis abstracta. A manera de recordatorio, colocamos aquí su sintaxis concreta:

```
<S> := <Expr>

<Expr> ::= <Int>
       | (+ <Expr> <Expr>)
       | (- <Expr> <Expr>)

<Int> := <N>
       | -<M>

<D> := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<N> := 0 | <D>{<N>}

<M> := <D>{<N>}
```

Por facilidad, para definir la sintaxis abstracta, nos enfocaremos únicamente en la parte de la gramática que indica cómo construir expresiones:

```
<Expr> ::= <Int>
       | (+ <Expr> <Expr>)
       | (- <Expr> <Expr>)
```

Tenemos entonces tres tipos de expresiones:

- Enteros, los cuales representaremos con la etiqueta *Num*. Al ser un valor atómico, únicamente tendrán como hijo al valor entero que representan.
- Sumas, las cuales etiquetaremos con la etiqueta *Add*. Como podemos apreciar de la gramática, las sumas requieren dos expresiones, por lo que se trata de un nodo con dos hijos.
- Restas, las cuales etiquetaremos con la etiqueta *Sub*. Al igual que las sumas, se tratará de nodos que tengan dos hijos.

Para definir las reglas que nos permitan especificar a los ASA, definimos la siguiente relación:

$$a \ \mathsf{ASA}$$

que se lee como *a es un* ASA. Dicho esto, definimos las reglas y colocamos una descripción de cuál sería su lectura.

Números
$Num(n)$ es un ASA si $n \in \mathbb{Z}$.

$$\frac{n \in \mathbb{Z}}{Num(n) \ \mathsf{ASA}}$$

Suma
$Add(i, d)$ es un ASA si tanto i como d son ASAs también. Es decir, es un árbol con dos subárboles.

$$\frac{i \ \mathsf{ASA} \qquad d \ \mathsf{ASA}}{Add(i, d) \ \mathsf{ASA}}$$

Resta
$Sub(i, d)$ es un ASA si tanto i como d son ASAs también.

$$\frac{i \ \mathsf{ASA} \qquad d \ \mathsf{ASA}}{Sub(i, d) \ \mathsf{ASA}}$$

Esta formalización permite además, escribir en formato plano los árboles sin necesidad de depender de su representación gráfica.

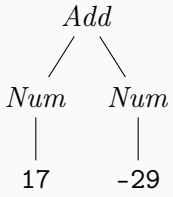
Ejemplo 2: ASA de una expresión en formato plano vs. representación gráfica

Usando la especificación del ejemplo anterior, veremos cómo obtener la sintaxis abstracta de la expresión que analizamos en la nota pasada: `(+ 17 -29)`

Representación en formato plano

$Add\,(Num\,(\mathtt{17})\,, Num(\mathtt{-29}))$

Representación en formato de árbol



3. De la sintaxis concreta a la abstracta

La relación entre la sintaxis concreta y la sintaxis abstracta puede definirse formalmente como la composición de las funciones de análisis léxico y sintáctico que mencionamos en la nota de clase anterior, donde la función `lexer` transforma el texto fuente en una secuencia de *tokens* y la función `parser` transforma esa secuencia de *tokens* en una estructura jerárquica como un ASA.

Definición formal

Dada una entrada de texto fuente S, el analizador léxico produce una secuencia de *tokens* `[Token]` utilizando una función de mapeo `lexer:` $S \rightarrow$ `[Token]`. Cada *token* en `[Token]` representa un elemento léxico del texto fuente, como palabras reservadas, identificadores, operadores, etc.

Luego, el analizador sintáctico toma la secuencia de *tokens* `[Token]` y la convierte en una estructura jerárquica, como un ASA, utilizando una función de mapeo `parser: [Token]` $\rightarrow$ `ASA`.

Por lo tanto, la relación entre la sintaxis concreta y la sintaxis abstracta, denotada como ϕ, se puede definir como la composición de las funciones `lexer` y `parser`:

$$\phi = \texttt{lexer} \circ \texttt{parser} : S \rightarrow \texttt{ASA}$$

Tomando como base nuestros ejemplos de la nota anterior y ésta. Tenemos como ejemplo que $\phi\,(\texttt{(+ 17 -29)} = \mathit{Add}\,(\mathit{Num}\,(\texttt{17})\,, \mathit{Num}(\texttt{-29}))$.

El laboratorio profundizará en cómo definir estas funciones.

4. Conclusión

En conclusión, el estudio de la sintaxis abstracta constituye un paso fundamental en la especificación formal de los lenguajes de programación. Al abstraer los detalles superficiales de la sintaxis concreta, el Árbol de Sintaxis Abstracta ofrece una representación clara, jerárquica y manipulable de los programas, lo que facilita el análisis, la optimización y la implementación de nuevas características en un lenguaje. Comprender esta distinción no solo fortalece el diseño y la eficiencia de compiladores e intérpretes, sino que también sienta las bases conceptuales para continuar con el estudio de las fases posteriores del proceso de traducción, donde la formalización se convierte en una herramienta indispensable para garantizar precisión, robustez y coherencia en el desarrollo de software.