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

Sintaxis Concreta

Manuel Soto Romero*

Semestre 2026-1
Facultad de Ciencias UNAM

Con esta nota damos inicio al desarrollo del tema Especificación Formal de Lenguajes de Programación. Recordemos que la especificación formal en la teoría de lenguajes de programación es de gran importancia porque nos permite asegurar la precisión, la corrección y la eficiencia tanto en el diseño como en la implementación y evolución de los lenguajes de programación.

En esta primera nota de la Unidad 2, dedicada a la especificación formal, comenzaremos a transitar desde la intuición hacia la formalidad. El objetivo es que se comprenda cómo podemos describir un lenguaje de manera rigurosa, sin ambigüedades y con bases matemáticas sólidas.

En particular, revisaremos algunos de los conceptos empleados en la formalización de la sintaxis de un lenguaje de programación. La sintaxis se refiere a las reglas que definen cómo se deben estructurar y organizar los símbolos y palabras reservadas del lenguaje para formar programas correctos. Estas reglas determinan qué combinaciones de caracteres y estructuras son aceptables y cuáles no.

En el contexto de la teoría de lenguajes de programación y lenguajes formales, la sintaxis concreta se refiere a la estructura específica de un lenguaje de programación que define exactamente cómo se deben escribir los programas. Matemáticamente, esto se describe mediante una gramática formal que especifica las reglas de formación para las secuencias válidas de símbolos en el lenguaje. Esta especificación formal suele dividirse en dos niveles: la sintaxis léxica y la sintaxis libre de contexto, que en conjunto permiten construir programas válidos y sin ambigüedades.

Para ejemplificar estos conceptos trabajaremos con un subconjunto del lenguaje Lisp, reducido y adaptado con fines pedagógicos, al que llamaremos MiniLisp. A lo largo del curso, este lenguaje nos servirá como hilo conductor: lo iremos extendiendo progresivamente para ilustrar distintos aspectos de diseño de lenguajes, desde la especificación de la sintaxis hasta la construcción de intérpretes y compiladores. En esta nota, comenzaremos con su sintaxis concreta, construida a partir de la sintaxis léxica y la sintaxis libre de contexto, y veremos cómo estas definiciones se formalizan mediante expresiones regulares, autómatas y gramáticas.

1. MiniLisp

Lisp (*LISt Processing*) es un lenguaje de programación de alto nivel diseñado inicialmente por John McCarthy en 1958 para el procesamiento de datos simbólicos. Lisp se caracteriza por su uso extensivo de *s-expressions* (expresiones simbólicas) y su enfoque en la manipulación de listas como estructuras de

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

datos fundamentales. Es conocido por su flexibilidad, su poderosa capacidad para la manipulación de datos y funciones, y su uso en Inteligencia Artificial. Es considerado el primer lenguaje de programación funcional de la historia.

Para ejemplificar los conceptos y decisiones de diseño de lenguajes, usaremos un subconjunto de Lisp que iremos extendiendo a lo largo del curso al cual llamaremos MiniLisp. Usaremos este pequeño lenguaje por varias razones pedagógicas y técnicas:

- **Simplicidad y uniformidad en sintaxis:** Tiene una sintaxis extremadamente simple y uniforme basada en *s-expressions*. Todo se representa como listas, lo que facilita el análisis y la manipulación del código. Esta uniformidad te ayudará a enfocarte en los conceptos de diseño y/o programación en lugar de preocuparte por la sintaxis del lenguaje.
- **Introducción a la programación funcional:** Lisp es uno de los lenguajes más antiguos que soporta programación funcional. Estudiar MiniLisp te permitirá comprender conceptos fundamentales de la programación funcional, como las funciones de primera clase, las funciones de orden superior, y la recursión.
- **Historia y contexto:** Lisp tiene una rica historia y ha influido profundamente en muchos lenguajes modernos (como Python, Haskell, Java y JavaScript). Conocer MiniLisp te proporcionará una comprensión del desarrollo de los lenguajes de programación y de cómo las ideas de este lenguaje han permeado otros lenguajes.
- **Árboles de sintaxis y compiladores:** Dado que el código de MiniLisp es esencialmente una estructura de datos (listas), es ideal para enseñar sobre árboles de sintaxis abstracta árboles de sintaxis abstracta (los estudiaremos en la nota siguiente) y otros conceptos introductorios de Compiladores. Podrás escribir intérpretes y compiladores para MiniLisp de una forma relativamente sencilla que te proporcionará experiencia práctica con estos conceptos.
- **Exploración de estilos:** MiniLisp soportará multiples estilos (funcional, estructurado y orientado a objetos), lo que te permitirá experimentar y comparar distintos estilos de programación dentro del mismo lenguaje.

MiniLisp Versión 1: Expresiones aritméticas

Para esta primera serie de notas, estudiaremos el diseño e implementación de expresiones aritméticas para MiniLisp. Dado que todo en este lenguaje se representa con *s-expressions,* empecemos con su definición y entendimiento.

Definición 1 (*s-expression*)

Definimos una ***s-expression*** *S de forma recursiva como sigue:*

- *Si $a \in \mathsf{ATOM}$, entonces $a \in S$.*
- *Si $e_1, e_2, \ldots, e_n \in S$ para $n \geq 1$ entonces $(e_1\ e_2\ \ldots\ e_n) \in S$.*
- *Son todas.*

Una *s-expression* es entonces una notación utilizada en MiniLsip para representar datos y código de manera uniforme y estructurada. Lo que nos dice la definición anterior es que una *s-expression* puede ser un átomo ($a \in \mathsf{ATOM}$) o una lista de *s-expressions* delimitada por paréntesis. Un átomo es una unidad indivisible, puede ser:

- Un símbolo: Una secuencia de caracteres que representa una variable, nombre de función, etc.
- Un número: Un entero, un número de punto flotantes, etc.
- Una cadena de texto: Una secuencia de caracteres entre comillas.

Para esta primera versión de MiniLisp nuestros átomos serán números enteros, y los operadores de suma (+) y resta (-). Lo cual nos permitirá generar *s-expressions* de la forma:

- `1729`
- `(+ 3 4)`
- `(- (+ 10 2) -8)`

Otro aspecto importante a considerar en MiniLisp, y que puede apreciarse en los ejemplos anteriores, es que usa notación prefija. Es decir, los operadores se colocan a la izquierda de los operandos.

De la informalidad a la formalidad

Esta forma de definir MiniLisp puede resultar confusa o en otras palabras ambigüa, por ejemplo: ¿qué es una lista? ¿cómo se escribe número de punto flotante? ¿qué es una variable? ¿cómo se escriben las funciones?

Esto es algo que pasa muy seguido cuando estamos aprendiendo a programar en un nuevo lenguaje de programación, es por ello que necesitamos formalizar este lenguaje, de forma tal que no haya lugar a duda cuando escribamos un programa correcto así como el significado de cada símbolo. Comenzaremos en esta nota con la formalización de su sintaxis.

2. Sintaxis léxica

La sintaxis léxica se refiere a la estructura de los *tokens* (lexemas) básicos de un lenguaje de programación, que son las unidades mínimas de significado, tales como palabras reservadas, identificadores, literales, operadores y delimitadores. Formalmente, la sintaxis léxica se define usando expresiones regulares y autómatas finitos.

Definición formal de sintaxis léxica

Para definir formalmente la sintaxis léxica, consideremos los siguientes componentes:

- **Alfabeto (Σ):** Un conjunto finito de símbolos o caracteres que se pueden utilizar en el lenguaje. Esto incluye letras, dígitos, y caracteres especiales.
- ***Tokens*:** Secuencias de caracteres que forman unidades léxicas básicas del lenguaje. Cada tipo de *token* puede ser descrito por una expresión regular.

Definición 1 (*Expresión regular*)

*Una **expresión regular** E sobre un alfabeto Σ se define recursivamente como:*

- *ε (la cadena vacía) es una expresión regular.*

- *Para cualquier símbolo $a \in \Sigma$, a es una expresión regular que representa la cadena que contiene solo a.*
- *Si E_1 y E_2 son expresiones regulares, entonces E_1E_2 (concatenación) es una expresión regular.*
- *Si E_1 y E_2 son expresiones regulares, entonces $E_1 + E_2$ (unión) es una expresión regular.*
- *Si E es una expresión regular, entonces $E^\star$ (cerradura de Kleene) es una expresión regular, representando cero o más repeticiones de E.*

Ejemplo 1: Sintaxis léxica de MiniLisp

Definamos la sintaxis léxica de nuestro lenguaje. Para ello, comenzamos especificando cuál es el alfabeto. Recordemos que el lenguaje. Para esta primera versión consideraremos que los átomos de las *s-expression* son únicamente números enteros. Recordemos también el uso de paréntesis y nuestros operadores de suma y resta. Con ello el alfabeto queda especificado por:

$$\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{-}, \texttt{+}, \texttt{(}, \texttt{)}\}$$

Ahora, los *tokens* de nuestro lenguaje serán justamente los símbolos o átomos que podemos formar con dichos símbolos. En este sentido tenemos los siguientes tipos de *tokens*:

- Paréntesis
- Operadores aritméticos
- Números enteros

Veamos las expresiones regulares para cada uno de ellos.

Paréntesis: Únicamente son dos posibilidades, el paréntesis que abre y el que cierra.

$$\texttt{(} + \texttt{)}$$

Operadores aritméticos: Únicamente son dos posibilidades, el operador de suma y el de resta.

$$\texttt{+} + \texttt{-}$$

Números enteros: Para generar números enteros, estableceremos algunos conjuntos de dígitos primero:

$$D = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}\}$$

$$Z = D - \{0\}$$

$$ZD^\star + \texttt{-}ZD^\star$$

Es decir, un número entero puede tener o no signo negativo y debe comenzar con cualquier dígito de cero seguido de cualquier combinación de dígitos. De esta forma números como `01` no son válidos en el lenguaje.

Las expresiones regulares, además, pueden ser convertidas en autómatas finitos deterministas (AFD o DFA) para su procesamiento. No estudiaremos como hacer esta conversión pues lo estudiaste en el curso de Autómatas y Lenguajes Formales y lo retomarás en el curso de Compiladores, por lo tanto omitiremos su explicación de momento.

Decimos entonces que la sintaxis léxica se define formalmente mediante expresiones regulares que describen cómo se forman los *tokens*. Esta estructura formal permite que los analizadores léxicos (*lexers*) procesen el código fuente y lo dividan en *tokens* que luego pueden ser procesados por el analizador sintáctico (*parser*). Hablaremos de esto en otra nota.

3. Sintaxis libre de contexto

La sintaxis libre de contexto se refiere a la estructura de un lenguaje de programación en la que las reglas de formación de sus sentencias se pueden describir mediante una gramática libre de contexto (GLC o CFG).

Definición 3 (*Gramática Libre de Contexto*)

Una ***gramática libre de contexto*** *se define como una 4-tupla* $G = (N, \Sigma, P, S)$*, donde:*

- N *es un conjunto finito de* ***símbolos no terminales*** *(o* ***variables****). Estos símbolos representan categorías sintácticas y pueden ser descompuestos en otras categorías o en símbolos terminales.*
- Σ *es un conjunto finito de* ***símbolos terminales*** *(o* ***constantes****). Estos son los símbolos básicos del lenguaje, que aparecen en las sentencias del lenguaje (por ejemplo, palabras reservadas, operadores, identificadores, etc.).*
- P *es un conjunto finito de* ***reglas de producción****. Cada regla de producción tiene la forma* $A \to \alpha$*, donde* $A \in N$ *y* $\alpha \in (N \cup \Sigma)^\star$ *.*
- S *es el* ***símbolo inicial****, tal que* $S \in N$ *desde el cual se empieza la derivación de las cadenas del lenguaje.*

Ejemplo 4: Sintaxis Libre de Contexto de MiniLisp

Las reglas de escritura para combinar los *tokens* en esta versión de nuestro lenguaje son simples:

- Toda expresión debe ir delimitada por paréntesis
- Se usa notación prefija
- Las operaciones son binarias

Con esto podemos definir entonces la gramática como sigue:

```
S → E

E → I
E → (+ E E)
E → (- E E)

I → N
I → -M
```

```
N → 0
N → DN
N → D

D → 1
...
D → 9

M → DN
```

Aunque esta forma de definir gramáticas tiene el beneficio de la formalidad para evitar ambigüedades al escribir expresiones de los lenguajes de programación, es difícil de leer y tediosa de escribir. Esto se debe a que la naturaleza de esta notación nació en el contexto de las matemáticas. Consientes de esto, entre los años 1950 y 1960, John Backus y Peter Naur inventan la notación BNF (*Backus-Naur Form*) la cual surgió como una solución a la necesidad de una manera más clara y precisa de definir la sintaxis de los lenguajes de programación.

Notación BNF

La notación BNF es una notación formal para describir la sintaxis de lenguajes formales. Una de sus características más importantes es la capacidad de definir alternativas mediante el uso del operador `|`.

Componentes básicos de BNF

Variables Se escriben generalmente entre diamantes `<>`.

Constantes Se representan igual que en la notación clásica.

Reglas de producción Se representan igual que en la notación clásica pero usando los símbolos `::=` en lugar de →.

Uso del operador `|`

El operador `|` se utiliza para especificar alternativas en las reglas de producción, evitando escribir un renglón nuevo para cada regla. Esto significa que la variable del lado izquierdo puede expandirse en cualquiera de las alternativas proporcionadas.

Ejemplo 5: Sintaxis Libre de Contexto de MiniLisp en BNF

```
<S> ::= <Expr>

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

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

```
<D> := 1 | ... | 9

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

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

Esta notación redujo significativamente la forma en que denotamos a los lenguajes de programación. Sin embargo, el muchas de las reglas recursivas generan confusión, mismas que en la práctica son propensas a ambigüedades y vienen del hecho de que es tedioso escribir una y otra vez reglas de producción similares.

Notación EBNF

Después de la introducción de BNF en los años 1960, se reconoció la necesidad de una notación más poderosa que pudiera describir la sintaxis de los lenguajes de programación de una manera más concisa y legible. Aunque BNF era efectiva, tenía limitaciones en términos de expresividad, especialmente para describir repeticiones y agrupaciones de una manera más compacta.

Entre los años 1970 y 1980, varixs investigadorxs y desarrolladorxs de lenguajes de programación comenzaron a proponer y utilizar extensiones de BNF para superar sus limitaciones. Una de las primeras formalizaciones de EBNF (*Extended Backus Naur Form*) fue realizada por Niklaus Wirth, un influyente científico de la computación suizo conocido por desarrollar los lenguajes Pascal y Modula-2. Wirth publicó su versión de EBNF en su libro *Algorithms + Data Structures = Programs* en 1976.

A lo largo de los años, varias organizaciones de estandarización, incluyendo ISO, adoptaron y formalizaron EBNF como un estándar para la descripción de lenguajes de programación. La ISO publicó el estándar ISO/IEC 14977 para EBNF en 1996.

Componentes básicos de EBNF

Repetición En EBNF las secuencias que pueden repetirse cero o más veces se indican usando llaves `{}`.

Opcionalidad Las secuencias opcionales se indican usando corchetes `[]`.

Agrupaciones Las agrupaciones de elementos se indican usando paréntesis `()`.

Alternativas Las alternativas se heredan de BNF mediante el operador `|`.

Ejemplo 6: Sintaxis Libre de Contexto de MiniLisp en EBNF

```
<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>}
```

Ejemplo 7: Derivación de una expresión en MiniLisp

Veamos cómo derivar la expresión `(+ 17 -29)` usando la gramática del Ejemplo 6:

```
<S> →
<Expr> →
(+ <Expr> <Expr>) →
(+ <Int> <Expr>) →
(+ <N> <Expr>) →
(+ <D>{<N>} <Expr>) →
(+ 1{<N>} <Expr>) →
(+ 17 <Expr>) →
(+ 17 <Int>) →
(+ 17 -<M>) →
(+ 17 -<D>{<N>}) →
(+ 17 -2{<N>}) →
(+ 17 -29)
```

4. Sintaxis concreta

La sintaxis concreta de un lenguaje de programación puede formalizarse matemáticamente utilizando las definiciones de la sintaxis léxica y la sintaxis libre de contexto.

Definición 3 (*Sintaxis concreta de un lenguaje de programación*)

*La **sintaxis concreta** se puede definir como un par (L, G) donde:*

- L es la definición léxica representada por el conjunto de expresiones regulares R.
- G es la gramática libre de contexto (N, Σ, P, S).

Podemos pensar en la sintaxis concreta como las secuencias de caracteres del alfabeto Σ que se convierten en programas válidos del lenguaje. Esto se logra mediante los pasos siguientes:

Análisis léxico Definimos una función léxica `lexer` : $\Sigma^\star \to$ [*Token*] que toma una cadena de caracteres y produce una secuencia de *token*s según las expresiones regulares R.

Análisis sintáctico Definimos una función sintáctica $\texttt{parser} : [\mathit{Token}] \to \mathit{AST}$ que toma una secuencia de *tokens* y produce un árbol de sintaxis abstracta (AST) según la gramática libre de contexto G. Si el programa no respeta las reglas de sintaxis, este árbol no puede ser construido.

Con estos pasos podemos pensar a la sintaxis concreta como la composición de estas dos funciones:

$$\texttt{parser} \circ \texttt{lexer} : \Sigma^\star \to \mathit{AST}$$

Profundizaremos más en estos temas en nuestra nota de clase siguiente.

5. Conclusión

En conclusión, el estudio de la sintaxis concreta nos permite dar el primer paso en la formalización de un lenguaje de programación: pasar de la intuición y la práctica cotidiana de escribir programas, a una descripción rigurosa que elimina ambigüedades y asegura claridad en el diseño. Al trabajar con MiniLisp hemos visto cómo la sintaxis léxica y la sintaxis libre de contexto se combinan para definir, de manera precisa, qué constituye un programa correcto. Este marco formal no solo es fundamental para comprender la teoría de lenguajes y la construcción de compiladores, sino que también nos proporciona una base sólida para reflexionar sobre las decisiones de diseño en cualquier lenguaje de programación. Con estas herramientas, podremos avanzar hacia la siguiente etapa: la sintaxis abstracta y su papel en la representación interna de los programas.

Referencias

[1] S. Krishnamurthi, *Programming Languages: Application and Interpretation*. 2007.
[2] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.
[3] M. Gabbrielli y S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2023.
[4] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.
[5] H. R. Nielson y F. Nielson, *Semantics with Applications: An Appetizer*. Springer, 2007.
[6] K. D. Lee, *Foundations of Programming Languages*. Springer, 2017.
[7] International Organization for Standardization e International Electrotechnical Commission, *Syntactic Metalanguage*, 1996.