**Sintaxis de Rust**

**1.1. Gramática Léxica**

**1.1.1. Elementos Léxicos**

<token> ->

<palabra reservada> |

<identificador> |

<constante> |

<literal de cadena> |

<punctuator>

<token de preprocesamiento> ->

<nombre de encabezado> |

<identificador> |

<número de preprocesador>|

<constante carácter> |

<literal de cadena> |

<punctuator> |

Otro carácter no espacio en blanco

**1.1.2. Palabras Reservadas**

<palabra reservada> -> una de

abstract alignof as asm async await break crate dbg default do elif else enum extern false fn for if impl in let loop match mod move mut never new not null on override pub ref return Self Self\_ static struct super tailrec trait true try typeof unsized unsafe use where while yield

**1.1.3. Identificadores**

<identificador> -> <no dígito> | <identificador> <no dígito>

| <identificador> <dígito>

<no dígito> -> uno de \_ a b c d e f g h i j k l m n o p q r s t u v

w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9

**1.1.4. Constantes**

<constante> ->

<constante entera> |

<constante real> |

<constante carácter> |

<constante enumeración>

**Constante Entera**

<constante entera> ->

<constante decimal> |

<constante octal> |

<constante hexadecimal>

<constante decimal> ->

<dígito no cero> |

<constante decimal> <dígito>

<dígito no cero> -> uno de 1 2 3 4 5 6 7 8 9

<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9

<constante octal> -> 0o | <constante octal> <dígito octal>

<dígito octal> -> uno de 0 1 2 3 4 5 6 7

<constante hexadecimal> ->

0x <dígito hexadecimal> |

<constante hexadecimal> <dígito hexadecimal>

<dígito hexadecimal> -> uno de 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

No existen sufijos para la constante entera en rust, aunque se puede especificar el tipo como uno de los siguientes: i8 u8 i16 u16 i32 u32 i64 u64 i128 u128 isize usize.

**Constante Real**

<constante real> ->

<constante fracción> <parte exponente>? |

<secuencia dígitos> <parte exponente>?

<constante fracción> ->

<secuencia dígitos>? . <secuencia dígitos> |

<secuencia dígitos> .

<parte exponente> ->

e <signo>? <secuencia dígitos> |

E <signo>? <secuencia dígitos>

<signo> -> uno de + -

<secuencia dígitos> ->

<dígito> | <secuencia dígitos> <dígito>

<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9

**Constante Carácter**

<constante carácter> -> '<carácter-c>' | '<secuencia de escape>'

<carácter-c> -> cualquiera excepto ' \

<secuencia de escape> ->

<secuencia de escape simple> |

<secuencia de escape octal> |

<secuencia de escape hexadecimal>

<secuencia de escape simple> -> uno de \' \" \? \\ \b \f \n \r \t \u{07}

<secuencia de escape octal> ->

\<dígito octal> |

\<dígito octal> <dígito octal> |

\<dígito octal> <dígito octal> <dígito octal>

<dígito octal> -> uno de 0 1 2 3 4 5 6 7

<secuencia de escape hexadecimal> ->

\x<dígito hexadecimal> |

\x <dígito hexadecimal> <dígito hexadecimal>

<dígito hexadecimal> -> uno de 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

**Constante Enumeración**

<constante enumeración> -> <identificador>

**1.1.5. Constantes Cadena**

<constante cadena> ->

"<secuencia caracteres-s>"

<secuencia caracteres-s> ->

<carácter-s> | <secuencia caracteres-s> <carácter-s>

<carácter-s> ->

cualquiera excepto " \ |

<secuencia de escape>

<secuencia de escape> ->

<secuencia de escape simple> |

<secuencia de escape octal> |

<secuencia de escape hexadecimal>

<secuencia de escape simple> -> uno de \' \" \? \\ \b \f \n \r \t \u{07}

<secuencia de escape octal> ->

\ <dígito octal> |

\ <dígito octal> <dígito octal> |

\ <dígito octal> <dígito octal> <dígito octal>

<dígito octal> -> uno de 0 1 2 3 4 5 6 7

<secuencia de escape hexadecimal> ->

\x <dígito hexadecimal> |

\x <dígito hexadecimal> <dígito hexadecimal>

<dígito hexadecimal> -> uno de 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

**1.1.6. Punctuators – Caracteres de Puntuación**

punctuator -> uno de

[ ] ( ) { } . ->

++ -- & \* + - ~ !

/ % << >> < > <= >= == != ^ | && ||

= \*= /= %= += -= <<= >>= &= ^= |=

? : ; ... , # ::

**1.1.7. Nombre de Encabezados**

<Cargo.toml inclusion> -> <nombre de paquete> = “<version>”

<nombre de paquete> -> <identificador>

<version> -> <constante decimal> . <constante decimal> . <constante decimal>

<.rs inclusion> -> extern crate <nombre de paquete>

*Las inclusiones en rust se realizan en el .rs y en un archivo aparte llamado Cargo.toml*

**1.1.8. Números de Preprocesador**

Rust no maneja una sintaxis de números de preprocesador. Existen macros pero no pueden usarse directamente para declarar un valor numérico como en C. Si pueden usarse para declarar una constante numérica global con la sintaxis propia de las macros.

**1.2. Gramática de Estructura de Frases**

**1.2.1. Expresiones**

<expresión> ->

<expresión de asignación> |

<expresión> , <expresión de asignación>

<expresión de asignación> ->

<expresión condicional> |

<operador asignación> <expresión de asignación>

<expresión condicional> ->

<expresión O lógico> |

<expresión O lógico> ? <expresión> : <expresión condicional>

<operador asignación> -> uno de

= = /= %= += −= <<= >>= &= ^= |=

<expresión O lógico> ->

<expresión Y lógico> |

<expresión O lógico> || <expresión Y lógico>

<expresión Y lógico> ->

<expresión O inclusivo> |

<expresión Y lógico> && <expresión O inclusivo>

<expresión O inclusivo> ->

<expresión O excluyente> |

<expresión O inclusivo> | <expresión O excluyente>

<expresión O excluyente> ->

<expresión Y> |

<expresión O excluyente> ^ <expresión Y>

<expresión Y> ->

<expresión de igualdad> |

<expresión Y> & <expresión de igualdad>

<expresión de igualdad> ->

<expresión relacional> |

<expresión de igualdad> == <expresión relacional> |

<expresión de igualdad> != <expresión relacional>

<expresión relacional> ->

<expresión de corrimiento> |

<expresión relacional> < <expresión de corrimiento> |

<expresión relacional> > <expresión de corrimiento> |

<expresión relacional> <= <expresión de corrimiento> |

<expresión relacional> >= <expresión de corrimiento>

<expresión de corrimiento> ->

<expresión aditiva> |

<expresión de corrimiento> << <expresión aditiva> |

<expresión de corrimiento> >> <expresión aditiva>

<expresión aditiva> ->

<expresión multiplicativa> |

<expresión aditiva> + <expresión multiplicativa> |

<expresión aditiva> - <expresión multiplicativa>

<expresión multiplicativa> ->

<expresión de conversión> |

<expresión multiplicativa> <expresión de conversión> |

<expresión multiplicativa> / <expresión de conversión> |

<expresión multiplicativa> % <expresión de conversión>

<expresión de conversión> ->

<expresión unaria> |

(<nombre de tipo>) <expresión de conversión>

<expresión sufijo> ->

<expresión primaria> |<expresión sufijo> [<expresión>] | /\* arreglo \*/

<expresión sufijo> (<lista de argumentos>?) | /\* invocación \*/

<expresión sufijo> . <identificador> |

<expresión sufijo> -> <identificador>

<lista de argumentos> ->

<expresión de asignación> |

<lista de argumentos> , <expresión de asignación>

<expresión primaria> ->

<identificador> |

<constante> |

<constante cadena> |

(<expresión>)

**1.2.2. Declaraciones**

<declaración> ->

<variable> (<declaración>)? |

<constante> (<declaración>)? |

<funcion> (<declaración>)? |

<estructura> (<declaración>)? |

<array> (<declaración>)?

<variable> ->

| let (mut)? <variable>(: <tipo entero>)? = <constante entera>;

| let (mut)? <variable>(: <tipo real>)? = <constante real>;

| let (mut)? <variable>(:char)? = <constante carácter>;

| let <variable>(:&str)? = <constante cadena>; //no puede modificarse

<constante> ->

| const <variable>(: <tipo entero>)? = <constante entera>;

| const <variable>(: <tipo real>)? = <constante real>;

| const <variable>(:char)? = <constante carácter>;

| const <variable>(: <&str>)? = <constante cadena>;

<variable> -> <identificador>

<funcion> -> fn <identificador>(<parametros>) (-><tipo>)? {declaración}

<parametros> -> <identificador>: <tipo>(, <parametros>)?

<tipo> -> uno de bool char i8 u8 i16 u16 i32 u32 i64 u64 i128 u128 isize usize f32 f64

<estructura> -> struct <identificador> { <elemento> }

<elemento> -> <identificador>: <tipo>, <elemento>

<array> -> let <identificador> : [<tipo>, <constante entera>] = [<constante>, <constante>, …]