Universidad Nacional de Cuyo

Facultad de Ingeniería

tinyRust+

Manual del Usuario

Autor: Morales, Juan Martín

Docente a cargo: Dra. Ana Carolina Olivera

Contenido

Contenido 2
1 Introducción 4
2 Características del lenguaje tinyRust+ 4
3 Conceptos básicos 5
3.1 Objeto 5
3.2 Clase 5
3.3 Miembros de una clase 6
3.4 Herencia 8
3.5 Variables y atributos 9
3.6 Tipos 9
3.6.1 Tipos de datos primitivos 9
3.7 Literales 10
3.7.1 Literales de enteros 10
3.7.2 Literales cadenas 10
3.7.3 Literales caracteres 10
3.7.4 Literal nulo 10
3.8 Expresiones 10
3.8.1 Expresiones aritméticas, de comparación y operadores lógicos 11
3.8.2 self 11
3.8.3 Llamadas a métodos 12
3.8.4 Creación de objetos y arrays 12
3.8.5 Acceso a atributos 12
3.9 Sentencias 13
3.9.1 Asignación 13
3.9.2 Sentencia simple 13
3.9.3 Bucle While 13
3.9.4 Bloques 13
3.9.5 Condicionales 14
3.9.6 Retorno de método 14

3.10 Lexemas y tokens 14
3.10.1 Espacios en blanco 14
3.10.2 Comentarios 15
3.10.3 Identificadores 15
3.10.4 Palabras clave 15
4 Clases Predefinidas 15
4.1.1 Clase predefinida Object 16
4.1.2 Clase predefinida IO 16
4.1.3 Clase Array 16
4.1.4 I32 16
4.1.5 Str 17
4.1.6 Bool 17
4.1.7 Char 17
5 Sintáxis 17
6 Ejemplo: Fibonacci 20

1 Introducción

El presente manual brinda información necesaria para comprender las características y sintáxis del lenguaje tinyRust+. Este es una versión reducida del popular Rust con ciertas modificaciones que simplifican el proceso de compilación.

En este documento, encontrarás una descripción detallada de las características de tinyRust+, junto con ejemplos de código que ayudarán la comprensión de estas características.

Además se brindará una guía para instalar el compilador y las herramientas necesarias. Junto a los pasos necesarios para compilar y ejecutar tu programa tinyRust+.

2 Características del lenguaje tinyRust+

El lenguaje tinyRust+ es un lenguaje de programación reducido del popular Rust, además tiene modificaciones en la estructura para facilitar el desarrollo. El alcance del proyecto es suficiente para desarrollar habilidades académicas, aunque también incluye características propias de los lenguajes modernos.

Es un lenguaje orientado a objetos con un tipado fuerte y estático, en el cual las variables tienen un tipo de dato asociado que es estrictamente aplicado en tiempo de compilación. Este sistema de tipos garantiza que no ocurran errores de tipo en tiempos de ejecución.

Al ser orientado a objetos el código de tinyRust+:

- Se organiza en clases que pueden heredar atributos y comportamientos de otras clases.
- Es posible sobrescribir los métodos no estáticos heredados, pero no los atributos.
- También permite polimorfismo en los objetos instanciados de las clases.
- Permite declarar la visibilidad de un atributo para cumplir con el principio de ocultamiento.

Cada código a compilar debe tener un punto de entrada, que en este caso es un método *main*, que deberá estar declarado fuera del *scope* clases.

A medida que se avance en las distintas etapas se detallarán en mayor medida las características de tinyRust+.

3 Conceptos básicos

Al ser tinyRust+ un lenguaje orientado a objetos, además de los conceptos tradicionales de un lenguaje de programación, hay un serie de conceptos adicionales que debes manejar para poder desarrollar un programa con este lenguaje.

3.1 Objeto

Es un elemento de software que intenta representar un objeto del mundo real. De esta forma un objeto tendrá sus propiedades y acciones a realizar con el objeto. Estas propiedades y acciones están encapsuladas dentro del objeto, cumpliendo así los principios de encapsulamiento.

Un objeto tiene su estado (o estados) y su comportamiento. Esto se modela mediante propiedades (o atributos) y métodos. Incluso un objeto puede contener a su vez a otro tipo de objeto.

El uso de objetos nos proporciona los siguientes beneficios:

- Modularidad, el objeto y sus propiedades puede ser pasado por diferentes estructuras del código fuente, pero el objeto es el mismo.
- Encapsular Datos, ocultamos la implementación de propiedades del objeto ya que accederemos a través de los métodos del objeto.
- Reutilización de Código, podemos tener diferentes instancias de un objeto de tal manera que esas diferentes instancias están compartiendo el mismo código.
- Reemplazo, podemos reemplazar un objeto por otro siempre y cuando estos objetos tengan el mismo comportamiento.

3.2 Clase

Las clases representan los prototipos de los objetos que tenemos en el mundo real. Es decir, es una generalización de un conjunto de objetos. A su vez los objetos serán instancias de una determinada clase. En la clase es dónde realmente definimos los atributos y métodos que podrán contener cada una de las instancias de los objetos.

Todo el código de tinyRust+ está organizado en clases. Múltiples clases pueden ser definidas en el mismo archivo. La definición de una clase tiene la siguiente forma:

```
class <IdClase> [ : <IdClase> ] {
      <Miembros>*
};
```

En tinyRust+ una clase que representa a un tríangulo se vería:

```
class Triangulo {
    I32: base;
    I32: altura;
    create(I32:base, I32:altura) {
        self.base = base;
        self.altura = altura;
    }
    fn area() -> I32 {
        return (base*altura)/2;
    }
}
```

Tiene dos atributos con acceso privado (se verá mas adelante los modificadores de acceso), un método constructor (***create***) que instanciará los objetos (también se verá más adelante), y un método que calcula el área del tríangulo.

3.3 Miembros de una clase

El cuerpo de una definición de clase consiste en una lista de definiciones de miembros. Un miembro es un atributo o un método. Un atributo de una clase A especifica una variable que es parte del estado de objetos de la clase A. Un método de una clase A es un procedimiento que puede manipular variables y objetos de clase A.

Todos los atributos tienen un alcance privado salvo aquellos que tengan la palabra reservada pub del identificador de clase de los atributos que serán solo accesibles a la clase. Una subclase no puede acceder a variables de una superclase a menos que sean públicos, tampoco puede redefinirlos. Es decir, la única forma de proveer acceso a los atributos no públicos de una clase es a través de sus métodos. Todos los métodos tienen alcance global. Aquellos métodos que tengan antes de la palabra fn la palabra clave `static` serán métodos de clase. No puede accederse a una variable de instancia en un contexto estático. Por ejemplo,

```
class Cuadrado {
    I32: base;
    pub I32: altura;
    create(I32:base, I32:altura) {
        self.base = base;
        self.altura = altura;
    }
```

```
    fn sonDistintos(Cuadrado: cuadrad) -> Bool {
        ...
    }

    static fn sonIguales(Cuadrado: cuadrad) -> Bool {
        ...
    }
}
...
fn main() {
        I32: b;
        Cuadrado: c;
        Cuadrado: d;
        c = new Cuadrado(4,2);
        d = new Cuadrado(4,2);
        b = c.altura; // Acceso correcto al atributo de la clase
        b = c.base; // Acceso incorrecto al atributo de la clase
}
```

Los nombres de los miembros deben comenzar con letra minúscula. Dentro de una clase no se pueden definir métodos con el mismo nombre ni atributos con el mismo nombre. Pero un método y un atributo pueden tener el mismo nombre.

Veamos a continuación un ejemplo de un archivo `lista.tr` que muestra un ejemplo del uso de métodos y atributos.

```
class Cons : Lista {
        pub I32: xcar;
        pub Lista: xcdr;
        fn isNil() -> Bool{
                return false;
        }
        create(I32: hd, List: xcdr){
                xcar = hd;
                self.xcdr = xcdr;
        }
}
```

En este ejemplo, la clase `Cons` tiene dos atributos `xcar` y `xcdr` y dos métodos `isNil` e `create`. Tener en cuenta que los tipos de atributos, así como los tipos de parámetros formales y los tipos de métodos de retorno, son declarados explícitamente por el programador. Dado el objeto `c` de la clase `Cons` y el objeto `l` de la clase `Lista`, podemos establecer los campos `xcar` y `xcdr` utilizando el método init: `c.create (1, l).`

Esta notación es el pasaje de mensajes típico de la orientación a objetos. Puede haber definiciones de métodos create en muchas diferentes clases. El mensaje busca la clase del objeto `c` para decidir qué método create invocar. Como la clase de `c` es `Cons`, se invoca el método create en la clase `Cons`. Dentro de la invocación, las variables `xcar` y `xcdr` se refieren a los atributos de `c`. La variable especial `self` se refiere al objeto que envió el mensaje, que, en el ejemplo, es `c`.

`new A` genera un objeto nuevo de clase `A`. Un objeto puede considerarse como una entrada de clase en la tabla de símbolos que tiene un elemento para cada uno de los atributos de la clase, así como enlaces a los métodos de la clase. Una llamada al método `create` es: `(new Cons(1,new Nil()))`. Este ejemplo crea una entrada de objeto de clase `Cons` e inicializa el `xcar` de la celda de `Cons` para que sea 1 y el `xcdr` para que sea un new `Nil`.

3.4 Herencia

La herencia es una forma de estructurar el software. Mediante la herencia podemos indicar que una clase hereda de otra. Es decir la clase extiende las capacidades (propiedades y métodos) que tenga y añade nuevas propiedades y acciones.

En nuestro ejemplo del tríangulo, este podría heredar de una clase polígono.

```
class Triangulo:Poligono {
    ...
}
```

Si tenemos una clase B que hereda de una clase A entonces B hereda los miembros de A. Se dice que la clase A es la superclase de B y B es la subclase de A. La semántica de B hereda de A es que B tiene todos los miembros definidos en A además de las suyas propias.

En el caso de que una superclase y una subclase definan el mismo nombre de método, entonces la definición dada en la subclase tiene prioridad.

Es ilegal redefinir los nombres de los atributos. Además, para la seguridad de los tipos, es necesario imponer algunas restricciones sobre cómo se pueden redefinir los métodos. Si una definición de clase no especifica una clase principal, entonces la clase hereda de `Object` de forma predeterminada. Una clase puede heredar solo de una sola clase (no se permite herencia múltiple). La relación superclase-subclase en las clases define un gráfico. Este gráfico puede no contener ciclos. Por ejemplo, si B hereda de A, entonces A no debe heredar de B. Además, si B hereda de A, entonces A debe tener una definición de clase en algún lugar del código fuente.

Debido a que tinyRust+ tiene una herencia única, se deduce que si se satisfacen ambas restricciones, el gráfico de herencia forma un árbol con Object como raíz.

3.5 Variables y atributos

Las variables son un espacio de memoria en el que guardamos un determinado valor. Los atributos son variables que se encuentran declaradas en una clase. tinyRust+ es un leguaje de tipado estático. Por lo cual todas las variables tendrán un tipo de dato (ya sea un tipo de dato primitivo o de tipo referencia) y un nombre de identificador.

Las variables las declararemos al principio del bloque (estrictamente) de un método, y siguen la siguiente estructura:

```
<type>: <id>;
```

Los atributos pueden tener un modificador de visibilidad, que permitirá que dicho atributo pueda ser accedido desde fuera de la clase. Tiene la siguiente estructura:

```
[pub] <type>: <id>;
```

Para el caso particular de un atributo arreglo la definición tiene el siguiente formato:

```
[pub] Array <type>: <id>;
```

Y para las variables arreglo la definición es similar pero sin el modificador de visibilidad [pub].

3.6 Tipos

En tinyRust+, toda clase es un tipo. Y de forma predeterminada tiene un conjunto de datos primitivos, útiles para la construcción de otros tipos de datos.Todas las variables se inicializan por defecto para contener valores del tipo apropiado.

3.6.1 Tipos de datos primitivos

En tinyRust+ hay una serie de tipos de datos que son primitivos del lenguaje. Estos son:

- I32: números enteros.
- Str: cadenas de caracteres, con un máximo de 32 bytes (31 caracteres + caracter final).
- Bool: valores booleanos true y false.
- Char: caracteres.

3.7 Literales

Un literal es la representación en código fuente del valor de un tipo, como un número o un string. Son aquellos que podemos asignar a las variables de tipo primitiva o realizar operaciones aritméticas, en el caso de los enteros.

3.7.1 Literales de enteros

Los enteros son cadenas no vacías de dígitos del 0 al 9.

3.7.2 Literales cadenas

Las cadenas o *strings* están encarrados entre doble comillas "…".

3.7.3 Literales caracteres

Un carácter puede ser:

- `'x'` donde x es cualquier caracter excepto la barra invertida (`\`), el salto de línea, o la comilla simple (`'`). El valor del literal es el valor del caracter x.
- `'\x'` donde x es cualquier caracter excepto n o t. El valor del literal es el valor del caracter x. Por ejemplo el literal `'\v'` tiene valor v. Por ejemplo comillas simple (`\'`), comillas dobles (`\"`) y backslash (`\\`).
- `'\t'`. El valor del literal es el valor del caracter Tab.
- `'\n'`. El valor del literal es el valor del caracter salto de línea.

Una cadena no puede contener EOF. Una cadena no puede contener el null (caracter `\0`). Cualquier otro caracter puede incluirse en una cadena.

3.7.4 Literal nulo.

El literal nulo se representa mediante la palabra reservada nil. Por defecto las referencias en `tinyRust+` toman el valor nil si no son inicializadas de igual manera se utiliza nil para los arreglos no inicializados.

3.8 Expresiones

Una expresión es un conjunto de variables, operadores, invocaciones a métodos y literales que se construyen para poder ser evaluadas obteniendo un resultado.

Las expresiones en `tinyRust+` son:

- Expresiones aritméticas
- self
- Expresiones de comparación

- Expresiones lógicas
- Llamadas a métodos
- Construcciones de objetos.
- Acceso a atributos.

3.8.1 Expresiones aritméticas, de comparación y operadores lógicos

Las operaciones aritméticas binarias de `tinyRust+` son: `+, -, *, /, %`. La sintáxis es:

`<expr1> <op> <expr2>;`

Para evaluar dicha expresión, primero se evalúa `<expr1>` y luego `<expr2>`. El resultado de la operación es el resultado de la expresión. Los tipos estáticos de las dos subexpresiones deben ser `I32`. El tipo estático de la expresión es `I32`.

Al no incluir representaciones para número reales, solo enteros, `tinyRust+` solo tiene división de enteros.

En tinyRust+ las expresiones booleanas formadas con los operadores lógicos: `&&` (and), `||` (or) y `!` (not) y aquellas formuladas utilizando los operadores relacionales `<, <=,==,>= y !=`.

La comparación por igual `==` es un caso especial. Si `<expr1>` o `<expr2>` tiene el tipo estático `I32`, `Char`, `Bool` o `Str`, el otro debe tener el mismo tipo estático. Cualquier otro tipo puede compararse libremente. En objetos no básicos, la igualdad simplemente verifica la igualdad de la referencia (es decir, si las direcciones de memoria de los objetos son las mismas).

3.8.2 self

Es una referencia explícita a la instancia del tipo (objeto) en el que ocurre. La estructura general sigue la siguiente sintaxis:

`self.<id>;`

Un ejemplo de esta refencia lo podemos ver a continuación.

```
class ClaseA {
    I32: atributo1;
    create(I32: atributo1) {
        self.atributo1 = atributo1;
    }
}
```

3.8.3 Llamadas a métodos

Las llamadas a un método en tinyRust+ suelen tener la siguiente forma:

```
<expr>.<id>(<expr>, ..., <expr>);
```

Este es el caso de una llamada dentro de una expresión. La llamada como sentencia será vista en la sección 3.9.

3.8.4 Creación de objetos y arrays

Una expresión new tiene la forma:

```
new <type>([parametros actuales)]);
```

Para el caso de los *arrays* la expresión new queda de la siguiente forma:

```
new <type>([Expresion]);
```

El valor es un objeto nuevo de la clase apropiada. Tenga en cuenta que el new realiza la reserva de espacio para la referencia al objeto y la inicialización de la clase <type>. Se llama al método create de la clase si es que existe.

El new para un *Array* implica que se reserva el tamaño especificado por el valor de un literal entero resultante de evaluar la expresión entre corchetes y sus elementos son inicializados con el valor por defecto según su tipo primitivo de los elementos del arreglo. Además, el tipo declarado para los elementos debe coincidir con el tipo de la sentencia new. Por ejemplo,

```
Array I32: a;
a = new I32[4*2];
```

reserva el espacio de 8 elemento enteros y los inicializa a cada uno con cero. Luego, al acceder a cualquier elemento de la posición i, $0 \leq i < 8$, se obtiene 0.

3.8.5 Acceso a atributos

En tinyRust+ el acceso a atributos *private* de una clase fuera de la misma se realiza a través de los métodos que tenga definida dicha clase. Si el atributo está en la misma clase no hace falta poner ninguna expresión particular.

No es posible redefinir atributos sea cual sea su visibilidad. Cuando se crea (new) un nuevo objeto de una clase, se deben inicializar todos los atributos heredados (públicos) y locales. Los atributos heredados se inicializan primero en orden de herencia comenzando con los atributos de la clase de ancestro más grande. Dentro de una clase determinada, los atributos se inicializan en el orden en que aparecen en el código fuente. Los atributos son privados salvo que se utilice la palabra pub en cuyo caso serán globales. Los atributos de superclases privados no son accesibles por su subclase.

3.9 Sentencias

Un programa se componen de un conjunto de sentencias que en conjunto determinan un comportamiento durante la ejecución. Al final de cada una de las sentencias encontraremos un punto y coma `;`.

Las sentencias en `tinyRust+` incluyen a las expresiones, asignaciones, declaraciones ya sea de atributos, variables o métodos, blucles, bloques, condicionales. Se presenta a continuación una pequeá descripción de cada una.

3.9.1 Asignación

La asignación tiene la forma (tenga en cuenta que una asignación es una expresión aunque la utilizaremos como sentencia):

`<LadoDerecho>=<expr>;`

El tipo del lado izquierdo (`<expr>`) de la asignación debe *matchear* con el tipo del lado derecho.

3.9.2 Sentencia simple

`tinyRust+` realiza llamadas a métodos y otras expresiones como una sentencia de la siguiente forma:

(`<Expresion>`);

3.9.3 Bucle While

Las sentencias de bucle nos van a permitir ejecutar un bloque de sentencias tantas veces como la condición establecida se cumpla.

Un bucle while en tinyRust+ respeta la siguiente forma:

`while <condicion> <sentencia>`

La condición se evalúa antes de cada iteración del ciclo. Si la condición es falsa, el bucle termina. Si la condición es verdadera, se evalúa el cuerpo del bucle y el proceso se repite. El condicional debe tener tipo `Bool`.

3.9.4 Bloques

Un bloque en tinyRust+ respeta la siguiente forma:

`{ <sentencia>* }`

Las sentencias se evalúan en orden de izquierda a derecha.

Los bloques en tinyRust+ se diferenciarán entre bloque de método (aquellos que contienen declaraciones de variables) y bloque de sentencias (bloques dentro de if-else, while, etc. que no tienen declaracionesde variables). Esto quedará más claro al observar la gramática que establece las reglas sintácticas.

3.9.5 Condicionales

Es una sentencia de decisión la cual permite tomar una decisión sobre que sentencia ejecutar dada una condición.

La sentencia de condición tiene la forma:

```
if <condicion>
<sentencia>
[else {<sentencia>}] //se ejecutan si la condición es falsa
```

La semántica de los condicionales es la utilizada normalmente. La condición se evalúa primero. Si es verdadera, se evalúa lo que está dentro del if. Si la condición es falsa, se evalúa el else. El else es opcional.

3.9.6 Retorno de método

El retorno de un método tiene la forma:

```
return <expr>;
```

Solo se utiliza si el tipo de retorno es distinto de void. Además, el tipo de la expresión de retorno debe matchear con el tipo declarado de retorno de la función.

3.10 Lexemas y tokens

Los lexemas y tokens de tinyRust+ además de las cosas que se deben ignorar en tiempo de compilación se detallan a continuación:

3.10.1 Espacios en blanco

El espacio en blanco consta de cualquier secuencia de caracteres: blancos (ascii 32), \n (nueva línea, ascii10), \r (retorno de carro, ascii 13), \t (tabulación, ascii 9), \v (tabulación vertical, ascii 11).

3.10.2 Comentarios

tinyRust+ tiene dos tipos de comentarios:

- /* comentario */ Comentarios multi-línea: Todos los caracteres desde /* hasta */ son ignorados.
- // Comentario simple: Todos los caracteres de desde // hasta el final de la línea son ignorados.

3.10.3 Identificadores

Los nombres de las variables locales, los parámetros formales de métodos, self y atributos de clase son identificadores. El identificador self puede ser referenciado, pero es un error asignar self o vincular self como un parámetro formal. También es ilegal tener atributos denominados self.

Las variables locales y los parámetros formales tienen alcance léxico. Los atributos son visibles solo en la clase en que se declaran a no ser que estén declarados como pub en cuyo caso serán accesibles por las clases que heradan de ella y podrán accederse en código fuera de la clase como atributo de una instancia de dicha clase. La vinculación de una referencia de identificador es el ámbito más interno que contiene una declaración para ese identificador, o al atributo del mismo nombre si no hay otra declaración. La excepción a esta regla es el identificador self, que está implícitamente vinculado en cada clase.

Los identificadores son cadenas (distintas de las palabras clave) que constan de letras (**no esta incluida la letra ñ)**, dígitos y el carácter de guión bajo o underscore. Los identificadores de tipo (Clase) comienzan con una letra mayúscula. Los identificadores de objetos comienzan con una letra minúscula. Los identificadores self y void son tratados especialmente por tinyRust+ al igual que la palabra Array. Los símbolos sintácticos especiales (por ejemplo, paréntesis, corchetes, operador de asignación, etc.) se verán directamente en la gramática.

3.10.4 Palabras clave

Las palabras claves de tinyRust+ son: class, else, false, if, return, while, true, nil, false, new, fn, static, pub, self. En tinyRust+ las palabras clave son sensibles a mayúsculas y minúsculas.

4 Clases Predefinidas

Es interesante que ninguna de las clases base posee método create aunque las clases base deben inicializarse correctamente de acuerdo a su semántica. tinyRust+ cuenta con un pequeño conjunto de clases predefinidas y clases bases.

4.1.1 Clase predefinida Object

`Object`: La superclase de todas las clases de tinyRust+ (al estilo de la clase *java.lang.Object* de *Java*). En `tinyRust+`, la clase `Object` no posee métodos ni atributos.

4.1.2 Clase predefinida IO

`IO`: Contiene métodos útiles para realizar entrada/salida.

- `static fn out string(Str: s)->void`: imprime el argumento.
- `static fn out i32(I32: i)->void`: imprime el argumento.
- `static fn out bool(Bool: b)->void`: imprime el argumento.
- `static fn out char(Char: c)->void`: imprime el argumento.
- `static fn out array(Array: a)->void`: imprime el argumento.
- `static fn in str()->Str`: lee una cadena de la entrada estándar, sin incluir un carácter de nueva línea.
- `static fn in i32()->I32`: lee un entero de la entrada estándar.
- `static fn in bool()->Bool`: lee un booleano de la entrada estándar.
- `static fn in Char()->Char`: lee un caracter de la entrada estándar.

Tanto `Object` como `IO` pueden ser utilizadas sin necesidad de ser heredadas. `tinyRust+` tiene otras cuatro clases básicas: (`Char, I32, Bool, Str`).

4.1.3 Clase Array

La clase arreglo proporciona listas de tamaño estático de elementos de tipos primitivos (`I32, Bool,Str y Char`). Tenga en cuenta que dos arreglos serán compatibles si el tipo de sus elementos es el mismo y el tamaño también. De otra manera serán incompatibles.

- `fn length()->I32.`
- `length` devuelve la longitud del parámetro `self`.

El acceso a los elementos dentro de una clase arreglo se hace a través de su índice, `a[i]` donde `a` es un arreglo e `i` es el índice que ubica un elemento del arreglo. Tenga en cuenta que un arreglo comienza su índice en la posición 0 y el índice `i` no debe superar el tamaño del arreglo `0 < i < a.lenght() − 1`. Un arreglo no inicializado tiene tamaño 0. Es un error heredar o redefinir `Array`.

4.1.4 I32

La clase `I32` proporciona números enteros. No hay métodos especiales para `I32`. La inicialización predeterminada para las variables de tipo `I32` es 0 (no `void`). Es un error heredar o redefinir `I32`.

4.1.5 Str

La clase `Str` proporciona cadenas. Se definen los siguientes métodos:

- `fn length()->I32. length` devuelve la longitud del parámetro `self`.
- `fn concat(Str: s)->Str`. El método concat devuelve la cadena formada al concatenar s después de self.
- `fn substr (I32: i, I32: l)->Str`. El método substr devuelve la subcadena de su parámetro `self` comenzando en la posición `i` con longitud `l`; las posiciones de cadena se numeran comenzando en 0. Se genera un error en tiempo de ejecución si la subcadena especificada está fuera de rango.

La inicialización predeterminada para las variables de tipo Str es `""` (nunca `void`!). Es un error heredaro redefinir `Str`.

4.1.6 Bool

La clase `Bool` brinda el `true` y `false`. La inicialización predeterminada para las variables de tipo `Bool` es false (no `void`). Es un error heredar o redefinir `Bool`.

4.1.7 Char

La clase `Char` proporciona caracteres. Deben estar entre `' '`. Ver sección sobre literales caracteres. Es un error heredar o redefinir `Char`.

5 Sintáxis

A continuación se muestra la gramática libre de contexto en formato *BNF extendido* para `tinyRust+`. Solo aquellos programas escritos sintácticamente en este formato serán válidos para el futuro compilador. Para que sea más amena la lectura de la gramática la misma está escrita en notación *BNF extendida*.

⟨program⟩ ::= ⟨Clase⟩* ⟨Metodo-Main⟩

⟨Metodo-Main⟩ ::= fn main () ⟨BloqueMetodo⟩

⟨Clase⟩ ::= class id ⟨Herencia⟩? { ⟨Miembro⟩* }

⟨Herencia⟩ ::= : ⟨Tipo⟩

⟨Miembro⟩ ::= ⟨Atributo⟩ | ⟨Metodo⟩ | ⟨Constructor⟩

⟨Constructor⟩ ::= create ⟨Argumentos-Formales⟩ ⟨Bloque-Metodo⟩

⟨Atributo⟩ ::= ⟨Visibilidad⟩? ⟨Tipo⟩: ⟨Lista-Declaracion-Variables⟩ ;

⟨Metodo⟩ ::= ⟨Forma-Metodo⟩? fn id ⟨Argumentos-Formales⟩ -> ⟨Tipo-Metodo⟩ ⟨Bloque-Metodo⟩

⟨Visibilidad⟩ ::= pub

⟨Forma-Metodo⟩ ::= static

⟨Bloque-Metodo⟩ ::= { ⟨Decl-Var-Locales⟩* ⟨Sentencia⟩* }

⟨Decl-Var-Locales⟩ ::= ⟨Tipo⟩: ⟨Lista-Declaracion-Variables⟩ ;

⟨Lista-Declaracion-Variables⟩::= id

| id , ⟨Lista-Declaracion-Variables⟩

⟨Argumentos-Formales⟩::= (⟨Lista-Argumentos-Formales⟩?)

⟨Lista-Argumentos-Formales⟩ ::= ⟨Argumento-Formal⟩ , ⟨Lista-Argumentos-Formales⟩

| ⟨Argumento-Formal⟩

⟨Argumento-Formal⟩ ::= ⟨Tipo⟩ : id

⟨TipoMetodo⟩ ::= ⟨Tipo⟩ | void

⟨Tipo⟩ ::= ⟨Tipo-Primitivo⟩ | ⟨Tipo-Referencia⟩ | ⟨Tipo-Arreglo⟩

⟨Tipo-Primitivo⟩ ::= Str | Bool |I32 | Char

⟨Tipo-Referencia⟩ ::= id

⟨Tipo-Arreglo⟩ ::= Array ⟨Tipo-Primitivo⟩

⟨Sentencia⟩ ::= ;

| ⟨Asignacion⟩;| ⟨Sentencia-Simple⟩;

| if (⟨Expresion⟩) ⟨Sentencia⟩

| if (⟨Expresion⟩) ⟨Sentencia⟩ else ⟨Sentencia⟩

| while (⟨Expresion⟩) ⟨Sentencia⟩

| ⟨Bloque⟩

| return ⟨Expresion⟩?;

⟨Bloque⟩ ::= { ⟨Sentencia⟩* }

⟨Asignacion⟩ ::= ⟨AccesoVar-Simple⟩ = ⟨Expresion⟩

| ⟨AccesoSelf-Simple⟩ = ⟨Expresion⟩

⟨AccesoVar-Simple⟩ ::= id ⟨Encadenado-Simple⟩* | id [⟨Expresion⟩]

⟨AccesoSelf-Simple⟩ ::= self ⟨Encadenado-Simple⟩*

⟨Encadenado-Simple⟩ ::= . id

⟨Sentencia-Simple⟩ ::= (⟨Expresion⟩)

⟨Expresion⟩ ::= ⟨ExpOr⟩

⟨ExpOr⟩ ::= ⟨ExpOr⟩ || ⟨ExpAnd⟩ | ⟨ExpAnd⟩

⟨ExpAnd⟩ ::= ⟨ExpAnd⟩ && ⟨ExpIgual⟩ | ⟨ExpIgual⟩

⟨ExpIgual⟩ ::= ⟨ExpIgual⟩ ⟨OpIgual⟩ ⟨ExpCompuesta⟩ | ⟨ExpCompuesta⟩

⟨ExpCompuesta⟩ ::= ⟨ExpAd⟩ ⟨OpCompuesto⟩ ⟨ExpAd⟩ | ⟨ExpAd⟩

⟨ExpAd⟩ ::= ⟨ExpAd⟩ ⟨OpAd⟩ ⟨ExpMul⟩ | ⟨ExpMul⟩

⟨ExpMul⟩ ::= ⟨ExpMul⟩ ⟨OpMul⟩ ⟨ExpUn⟩ | ⟨ExpUn⟩

⟨ExpUn⟩ ::= ⟨OpUnario⟩ ⟨ExpUn⟩ | ⟨Operando⟩

⟨OpIgual⟩ ::= == | !=

⟨OpCompuesto⟩ ::= < | > | <= | >=

⟨OpAd⟩ ::= + | -

⟨OpUnario⟩ ::= + | - | !

⟨OpMul⟩ ::= * | / | %

⟨Operando⟩ ::= ⟨Literal⟩ | ⟨Primario⟩ ⟨Encadenado⟩?

⟨Literal⟩ ::= nil | true | false | intLiteral | StrLiteral | charLiteral

⟨Primario⟩ ::= ⟨ExpresionParentizada⟩

| ⟨AccesoSelf ⟩

| ⟨AccesoVar⟩

| ⟨Llamada-Metodo⟩

| ⟨Llamada-MetodoEstatico⟩

| ⟨Llamada-Constructor⟩

⟨ExpresionParentizada⟩ ::= (⟨Expresion⟩) ⟨Encadenado⟩?

⟨AccesoSelf ⟩ ::= self ⟨Encadenado⟩?

⟨AccesoVar⟩ ::= id ⟨Encadenado⟩? | id [⟨Expresion⟩]

⟨Llamada-Metodo⟩ ::= id ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩?

⟨Llamada-Metodo-Estatico⟩ ::= id . ⟨Llamada-Metodo⟩ ⟨Encadenado⟩?

⟨Llamada-Constructor⟩ ::= new id ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩?

| new ⟨Tipo-Primitivo⟩ [⟨Expresion⟩]

⟨Argumentos-Actuales⟩ ::= (⟨Lista-Expresiones⟩?)

⟨Lista-Expresiones⟩ ::= ⟨Expresion⟩ | ⟨Expresion⟩ , ⟨Lista-Expresiones⟩

⟨Encadenado⟩ ::= . ⟨Llamada-Metodo-Encadenado⟩

| . ⟨Acceso-Variable-Encadenado⟩

⟨Llamada-Metodo-Encadenado⟩ ::= id ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩?

⟨Acceso-Variable-Encadenado⟩ ::= id ⟨Encadenado⟩?

| id [⟨Expresion⟩]

6 Ejemplo: Fibonacci

```
class Fibonacci {
        I32: suma;
        I32: i,j;
        fn sucesion_fib(I32: n) -> I32{
                i=0; j=0; suma=0;
                while (i<=n) {
                        if (i==0){
                                (imprimo_numero(i));
                                (imprimo_sucesion(suma));
                        }
                        else if(i==1){
                                (imprimo_numero(i));
                                suma=suma+i;
                                (imprimo_sucesion(suma));
                        }
                        else {
                                (imprimo_numero(i));
                                suma=suma+j;
                                j=suma;
                                (imprimo_sucesion(suma));
                        }
                        i = i + 1;
                }
        }
        create(){
                i=0;
                j=0;
                suma=0;
        }
        fn imprimo_numero(I32: num) -> void{
                (IO.out_str("f_"));
                (IO.out_int(num));
                (IO.out_str("="));
        }
        fn imprimo_sucesion(I32: s) -> void{
                //"el valor es: ";
                (IO.out_i32(s));
                (IO.out_str("\n"));
        }
}
fn main(){
        Fibonacci: fib;
        I32: n;
        fib = new Fibonacci();
        n = IO.in_i32();
        (fib.sucesion_fib(n));
}
```