Compilador TinyRust+
Manual de Usuario

Autores:
Fernández, Juan Manuel
Volman, Mariel

Dra. Ana Carolina Olivera
Compiladores
1er semestre del 2023

CONTENIDO

1. Introducción..... 3
a. Características generales..... 3
b. Sintáxis básica..... 3
2. Declaraciones..... 4
a. Clases..... 4
b. El método main..... 4
c. Miembros..... 4
d. Herencia..... 6
e. Tipos..... 7
f. Definición de Métodos..... 7
g. Constructor..... 9
h. Declaración de atributos..... 9
i. Declaración de parámetros formales..... 9
3. Expresiones..... 10
a. Expresiones aritméticas, de comparación y operadores lógicos..... 10
b. self..... 10
c. Llamadas a métodos..... 11
d. New..... 11
e. Acceso a Atributos..... 11
4. Sentencias..... 12
a. Asignación..... 12
b. Sentencia Simple..... 12
c. Bucle while..... 13
d. Bloques..... 13
e. Condicionales..... 13
f. Retorno de método..... 14
5. Clases Base y Clases predefinidas..... 14
a. Clase predefinida Object..... 14
b. Clase Predefinida IO..... 14
c. Clase Array..... 15
d. I32..... 15
e. Str..... 16
f. Bool..... 16
g. Char..... 16
6. Lexemas y tokens en TinyRust+..... 17
a. Espacios en blanco..... 17
b. Comentarios..... 17
c. Identificadores..... 17
d. Palabras clave..... 18
e. Literales..... 18
i. Literales enteros..... 18

ii. Literales cadenas..... 18
iii. Literales caracteres..... 18
iv. Literales booleanos.....19
v. Literal nulo.....19
7. Gramática BNF extendida..... 19
a. Ejemplo.....22
8. Decisiones de diseño..... 23
9. Haciendo uso del compilador.....25

Licenciatura en Ciencias de la Computación
Universidad Nacional de Cuyo

1. Introducción

En el presente manual se detalla el modo de uso del compilador desarrollado durante el primer semestre de la asignatura “Compiladores” para un lenguaje de programación reducido llamado TinyRust+.

a. Características generales

TinyRust+ es un lenguaje de programación reducido basado en Rust, que cuenta con algunas limitaciones para facilitar el desarrollo del compilador en un semestre sin la utilización de herramientas automáticas. Aunque es un lenguaje académico incluye características propias de los lenguajes de programación modernos.

Todo el código de TinyRust+ está organizado en clases.

TinyRust+ es type safe: se debe garantizar que los métodos sean aplicados a datos de tipo correcto. Mientras que el tipado estático impone una disciplina fuerte de programación en TinyRust+, también garantiza que no ocurran errores de tipo en tiempo de ejecución.

TinyRust+ es fuertemente tipado.

b. Sintáxis básica

Un código fuente a compilar en TinyRust+ estará compuesto por un único archivo. El compilador de TinyRust+ compila un programa de la siguiente manera:

```
java -jar tinyRust.jar archivo_entrada.rs [archivo_salida]
```

El compilador compila el archivo “archivo_entrada.rs”. El nombre del archivo de salida es opcional. Si no se proporciona nombre para el archivo de salida el mismo es por defecto “archivo_entrada”. Tenga en cuenta que, aunque la extensión es *.rs* no tiene exactamente la misma sintaxis que el lenguaje Rust, por lo que dará error si intenta compilar con un compilador de Rust.

2. Declaraciones

a. Clases

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>*
};
```

Todos los nombres de clases son visibles globalmente (no hay clases privadas). Los nombres de Clases deben comenzar con una letra mayúscula. No puede haber dos clases con el mismo nombre en el mismo código fuente.

b. El método main

Cada código fuente debe tener un **único** método *main* que no tome parámetros formales.

El método *main* debe estar definido después de todas las definiciones de clases. Un programa en TinyRust+ comienza ejecutando el método *main*. No puede haber ninguna variable denominada *main* ya que es una palabra reservada del lenguaje.

```
fn main () {
     // Punto de entrada del programa
}
```

c. Miembros

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*. Los atributos privados serán solo accesibles a la clase que los contiene. Es decir, la única forma de proveer acceso a los atributos no públicos de una clase es a través de sus métodos. Una subclase no puede redefinir los atributos de su superclase.

Todos los métodos tienen un 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 Mundo {
      I32: a;
      pub Str: b;
       fn get_a()-> I32 { return a; }
       create(){ a = 42;}
       static fn imprimo_algo()-> void {
            IO.out_string("hola mundo");
       }
}
...
class Prueba{
...
Mundo: c;
c = new Mundo();
y = c.b; // Acceso correcto al atributo de la clase (alcance público)
z = c.a; // Acceso incorrecto al atributo de la clase (alcance privado)
c.imprimo_algo();
...
}
```

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 si pueden tener el mismo nombre.

Veamos a continuación un ejemplo de un archivo “lista.rs” que muestra un ejemplo del uso de métodos y atributos.

```
class Cons : Lista {
      pub I32: xcar;
      pub Lista: xcdr;
       fn isNil() -> Bool { 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 clases diferentes. 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 envía 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[1].

No hay ningún mecanismo en TinyRust+ para que los programadores desaloquen objetos. TinyRust+ tiene gestión automática de memoria. Los objetos que no van a ser utilizados por el programa son desalcados por el recolector de basura (garbage collector) en tiempo de ejecución.

d. Herencia

La herencia de clases está permitida. La herencia se define de la siguiente manera:

```
class B : A {
...
}
```

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 los suyos propios.

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 sólo de

[1] En este ejemplo Nil se asume como un subtipo de Lista

una clase (no se permite herencia múltiple). La relación superclase-subclase en las clases define un gráfico. Este gráfico no puede 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.

e. Tipos

En TinyRust+, toda clase también es un tipo. Una declaración de tipos tiene la forma `A:x`, donde *x* es una variable de tipo **A**. Toda variable debe tener una declaración de tipo en el punto en que aparece, por ejemplo, como parámetro formal en un método. Los tipos de todos los atributos deben estar también declarados.

La regla de tipo básica en TinyRust+ es que si un método o variable espera un valor de tipo **A**, entonces cualquier valor del tipo **B** se puede utilizar en su lugar, siempre que **A** sea un antepasado de **B** en la jerarquía de clases.

El sistema de tipos garantiza en tiempo de compilación que la ejecución de un programa no resulte en un error de tipo en tiempo de ejecución. Utilizando los tipos declarados para los identificadores en el código fuente el análisis semántico infiere un tipo para toda expresión en el código fuente.

Es importante distinguir entre el tipo asignado por el verificador de tipos para una expresión en tiempo de compilación que llamaremos tipo estático de la expresión y los tipos o tipo que la expresión puede evaluar durante la ejecución que llamamos tipos dinámicos.

La distinción entre tipo estático y tipo dinámico es necesaria debido a que el verificador de tipos no puede, en tiempo de compilación, tener información perfecta sobre qué valor se va a computar en tiempo de ejecución. De hecho, el tipo estático y el tipo dinámico suelen ser distintos. Lo que necesitamos, sin embargo, es que los tipos estáticos del verificador de tipos sean correctos con respecto a los tipos dinámicos.

Todas las variables en TinyRust+ se inicializan por defecto para contener valores del tipo apropiado. Se cuenta además con el tipo especial *void*.

f. Definición de Métodos

La definición de un método tiene la forma:

```
[ static ] fn <idMetodo>([<parametros formales>]) -> <type> { <sentencias> };
```

En caso de que no devuelva nada el tipo de retorno es *void*.

La declaración de métodos de clase estará precedida por la palabra reservada *static*. Si static no aparece se trata de una declaración de método de instancia.

Puede haber cero o más parámetros formales. Los identificadores utilizados en la lista de parámetros formales deben ser distintos. El tipo del cuerpo del método debe ajustarse al tipo de retorno declarado. Cuando se invoca un método, los parámetros formales se vinculan a los argumentos reales y se evalúa la expresión, el valor resultante es el significado de la invocación del método. Un parámetro formal oculta cualquier definición de un atributo del mismo nombre.

Para garantizar la seguridad de los tipos, existen restricciones sobre la redefinición de métodos heredados. La regla es simple: si una clase **B** hereda un método *f* de una superclase **A**, entonces **B** puede anular la definición heredada de *f* siempre que el número de argumentos, los tipos de parámetros formales y el tipo de retorno sean exactamente iguales en ambas definiciones. Tenga en cuenta que un método de clase no se puede redefinir. Para ver por qué es necesaria alguna restricción en la redefinición de métodos heredados, considere el siguiente ejemplo:

```
class A {
      fn f() -> I32 { return 1 };
};
class B : A {
      fn f() -> Str { return "1" };
};
```

Tomemos un objeto **a** con tipo dinámico **A**. Entonces, `a.f()+1` es una expresión bien formada con valor 2.

Sin embargo, no es posible sustituir un valor de tipo **B** para **a**, ya que trataríamos de sumar un número a un String. Luego, si los métodos pudieran redefinirse de forma arbitraria, entonces las subclases no podrían extender de forma sencilla el comportamiento de sus superclases.

La definición del método *create* es llamada únicamente cuando un nuevo objeto es inicializado. La definición de *create* tiene la siguiente forma:

```
create(<type>: <id>,...,<type>: <id>) { <sentencias> };
```

g. Constructor

create es la única función que puede tener argumentos distintos en una subclase y una superclase. Una expresión de inicialización provee acceso al inicializador de un tipo.

```
class UnaSubclase : UnaSuperClase {
    create() {
       // La inicialización de la subclase se coloca aquí
    }
}
```

Se llama a *create* al hacer *new* de una nueva instancia de clase. Tener en cuenta que *create* solo es utilizada al hacer *new*, no puede accederse en otro contexto.

h. Declaración de atributos

La definición de un atributo de clase tiene la siguiente forma:

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

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

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

Los atributos no pueden ser redefinidos.

i. Declaración de parámetros formales

En cuanto a la declaración de los parámetros formales esta debería ser de la forma:

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

En el caso de que el parámetro formal sea un arreglo la declaración es:

```
Array <type>: <id>
```

3. Expresiones

Las expresiones en TinyRust+ son:

a. Expresiones aritméticas, de comparación y operadores lógicos

TinyRust+ tiene cinco operaciones aritméticas binarias: +, -, *, / y %. La sintaxis 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.

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).

b. self

self es una referencia explícita a la instancia del tipo en el que ocurre.

```
self.id
```

Por ejemplo:

```
class UnaClase {
	Str: greeting;
	create(Str: greeting) {
		self.greeting = greeting;
	}
}
```

c. Llamadas a métodos

Las llamadas a un métodos en TinyRust+ suelen tener la siguiente forma:

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

Tenga en cuenta que en TinyRust+ los métodos siempre se llaman con los paréntesis, aunque no tengan parámetros. Este es el caso de una llamada dentro de una expresión. La llamada como sentencia será vista en la Sección 4.

d. New

Una expresión *new* tiene la forma `new <type>([<parámetros actuales>])`. Para el caso de los arreglos la expresión *new* queda de la siguiente forma: `new <type> [<Expresión>]`.

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[5+1];
```

reserva el espacio de 6 elementos enteros y los inicializa a cada uno con cero. Luego, al acceder al elemento a[2] se obtiene 0.

e. Acceso a Atributos

En TinyRust+ el acceso a atributos privados 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.

```
// Declaración de atributo público
     pub I32 : a;
// Declaración de atributo privado
     I32 : b;
```

4. Sentencias

Las sentencias en TinyRust+ incluyen a las expresiones, asignaciones, declaraciones de constantes, atributos y métodos, bucles, bloques y condicionales. Se presenta a continuación una breve introducción de cada uno.

a. Asignación

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

```
<LadoIzquierdo> = <expr>;
```

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

b. Sentencia Simple

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

```
(<Expresión>);
```

c. Bucle while

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**.

d. 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 declaraciones de variables).

e. Condicionales

La condición tiene la forma:

```
if (<condición>)
     <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 la sentencia dentro del *if*. Si la condición es **falsa**, se evalúa el *else*. El *else* es opcional.

f. 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 corresponderse con el tipo declarado de retorno de la función.

5. Clases Base y 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.

a. 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.

b. 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_string()->Str:` lee una cadena de la entrada estándar, sin incluir un caracter 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** y **Str**.

c. Clase Array

La clase **Array** proporciona listas de tamaño estático de elementos de tipos primitivos (I32, Bool, Str y Char). Tenga en cuenta que dos arreglos serían compatibles si el tipo de sus elementos es el mismo y el tamaño también. De otra manera serán incompatibles. Esta clase contiene un método *length* que devuelve la longitud del parámetro *self*.

```
fn length() -> I32 {}
```

El acceso a los elementos dentro de una clase **Array** 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**.

El siguiente código demuestra la creación y acceso a arreglos:

```
Array I32: a; // Declaramos un Array de tipo I32
fn nuevo_arreglo() -> void {
      // Inicializamos el Array de tamaño 3
      a = new I32[3];
      // Accedemos a la primera posición del Array
      a[0] = 1;
}
```

d. 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**.

e. Str

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

`fn length()->I32.` El método *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 heredar o redefinir **Str**.

f. 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**.

g. Char

La clase Char proporciona caracteres. Deben estar entre *' '*. Es un error heredar o redefinir **Char**.

> **Aclaración:** TinyRust+ solo provee las características indicadas en la sección precedente. Características no indicadas, tales como por ejemplo: genericidad, anotaciones, break, clausura, selectores, expresión implícita de miembros, tipos primitivos de punto flotante, etc., no están soportadas por el lenguaje. Y no se deben hacer puesto que NO serán válidas.

Las secciones restantes de este manual proporcionan una definición más formal de TinyRust+. La estructura léxica se abarca en la Sección 6 y la gramática generadora de código fuente en la Sección 7.

6. Lexemas y tokens en TinyRust+

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

a. Espacios en blanco

El espacio en blanco consta de cualquier secuencia de caracteres: blancos (ascii 32), \n (nueva línea, ascii 10), \r (retorno de carro, ascii 13), \t (tabulación, ascii 9), \v (tabulación vertical, ascii 11).

b. Comentarios

TinyRust+ tiene dos tipos de comentarios:

- **Comentarios multilínea**: Todos los caracteres desde /* hasta */ son ignorados.

```
/* comentario multilínea */
```

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

```
// Comentario simple
```

c. 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 heredan 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, dígitos y el caracter de guión bajo. 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.

d. Palabras clave

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

e. Literales

Un literal es la representación en código fuente del valor de un **tipo primitivo** (I32, Str, Char, Bool) o un literal nulo.

i. Literales enteros

Los enteros son cadenas no vacías de dígitos del 0 al 9. Por ejemplo: 42

ii. Literales cadenas

Las cadenas o Str están encerradas entre doble comillas "...". Por ejemplo: “hola mundo”

iii. 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, comillas simples (\'), comillas dobles (\") y barra invertida (\\)
- **'\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.

iv. Literales booleanos

El literal booleano puede ser **true** o **false**.

v. 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.

7. GRAMÁTICA BNF EXTENDIDA

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. Observe que en letra **negrita** aparecen los tokens devueltos por el Analizador Léxico.

⟨program⟩ ::= ⟨Clase⟩∗ ⟨Metodo-Main⟩

⟨Metodo-Main⟩ ::= **fn main ()** ⟨BloqueMétodo⟩

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

⟨Herencia⟩ ::= **:** ⟨Tipo⟩

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

⟨Constructor⟩ ::= **create** ⟨Argumentos-Formales⟩ ⟨Bloque-Método⟩

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

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

⟨Visibilidad⟩ ::= **pub**

```
⟨Forma-Método⟩ ::= static

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

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

⟨Lista-Declaración-Variables⟩::= id | id , ⟨Lista-Declaración-Variables⟩

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

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

⟨Argumento-Formal⟩ ::= ⟨Tipo⟩ : id

⟨Tipo-Método⟩ ::= ⟨Tipo⟩ | void

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

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

⟨Tipo-Referencia⟩ ::= id_clase

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

⟨Sentencia⟩ ::= ;
       | ⟨Asignación⟩ ;
       | ⟨Sentencia-Simple⟩ ;
       | if ( ⟨Expresión⟩ ) ⟨Sentencia⟩
       | if ( ⟨Expresión⟩ ) ⟨Sentencia⟩ else ⟨Sentencia⟩
       | while ( ⟨Expresión⟩ ) ⟨Sentencia⟩
       | ⟨Bloque⟩
       | return ⟨Expresión⟩? ;

⟨Bloque⟩ ::= { ⟨Sentencia⟩∗ }

⟨Asignación⟩ ::= ⟨AccesoVar-Simple⟩ = ⟨Expresión⟩
                | ⟨AccesoSelf-Simple⟩ = ⟨Expresión⟩

⟨AccesoVar-Simple⟩ ::= id ⟨Encadenado-Simple⟩∗ | id [ ⟨Expresión⟩ ]

⟨AccesoSelf-Simple⟩ ::= self ⟨Encadenado-Simple⟩∗
```

⟨Encadenado-Simple⟩ ::= **.** **id**

⟨Sentencia-Simple⟩ ::= **(** ⟨Expresión⟩ **)**

⟨Expresión⟩ ::= ⟨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-Método⟩
| ⟨Llamada-MétodoEstático⟩
| ⟨Llamada-Constructor⟩

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

⟨AccesoSelf⟩ ::= self ⟨Encadenado⟩?

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

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

⟨Llamada-Método-Estático⟩ ::= id . ⟨Llamada-Método⟩ ⟨Encadenado⟩?

⟨Llamada-Constructor⟩ ::= new id ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩?
                        | new ⟨Tipo-Primitivo⟩ [ ⟨Expresión⟩ ]

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

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

⟨Encadenado⟩ ::= . ⟨Llamada-Método-Encadenado⟩
             | . ⟨Acceso-Variable-Encadenado⟩

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

⟨Acceso-Variable-Encadenado⟩ ::= id ⟨Encadenado⟩? | id [ ⟨Expresión⟩ ]
```

a. Ejemplo

A continuación, se muestra un ejemplo de código escrito en TinyRust+.

```
class Fibonacci {
    pub I32: suma;
    I32: i,j;
    fn sucesion_fib(I32: n)-> void{
        I32: idx;
        i=0; j=1; suma=0; idx = 0;
        while (idx <= n){
            (out_idx(idx));
            (out_val(i));
            suma = i + j;
            i = j;
            j = suma;
            idx = idx + 1;
```

```
            }
        }
        create(){
            i=0;
            j=0;
            suma=0;
        }
        fn out_idx(I32: num) -> void{
            (IO.out_string("f_"));
            (IO.out_i32(num));
            (IO.out_string("="));
        }
        fn out_val(I32: s) -> void{
            (IO.out_i32(s));
            (IO.out_string("\n"));
        }
    }
    fn main(){
        Fibonacci: fib;
        I32: n;
        fib = new Fibonacci();
        // n = IO.in_i32();
    n = 12;
        (fib.sucesion_fib(n));
    }
```

8. Decisiones de diseño

Cada uno de los tipos primitivos reserva un total de 4 bytes (32 bits) de memoria, lo que implica la capacidad de utilizar enteros hasta 2,147,483,647 y -2,147,483,647, para el caso de las constantes strings, no tienen un límite impuesto más allá del máximo que se le permita al programa una vez cargado por el sistema operativo.

- Ejemplo de uso de strings:

```
fn main(){
    Str : hola;
    hola = "Hola mundo";
    (IO.out_string(hola));
}
```

- Ejemplo de uso de enteros válido:

```
fn main(){
    I32 : a;
    a = 2147483647;
    (IO.out_i32(a));
}
```

- Ejemplo de uso inválido:

```
fn main(){
    I32 : a;
    a = 2147483648;
    (IO.out_i32(a));
}
```

La decisión de limitar los enteros a este número es debido a que el modelo de memoria del set de instrucciones destino (MIPS) permite alocar de forma trivial dicha cantidad.

Por otro lado, las constantes strings, por ser eso mismo, constantes, pueden ser almacenadas de forma sencilla independientemente de su tamaño y por tanto, se da libertad al programador de utilizar las longitudes que le sean necesarias para sus cadenas.

9. Haciendo uso del compilador

Para comenzar el proceso de compilación, se requiere tener java 17 en adelante.

Una vez que se tienen los requerimientos necesarios, el compilador se invoca mediante la herramienta “jar” de java, el comando a ejecutar es “java -jar tinyRust+.jar <codigofuente.rs>”, lo cual tiene como salida el código ensamblador resultante del programa y las representaciones intermedias en formato json, que son la tabla de símbolos y el árbol de análisis sintáctico abstracto.

Cabe destacar que el archivo de entrada debe tener la extensión .rs y que bajo cualquier error léxico, sintáctico o semántico, el código ensamblador no será generado, en su lugar, se mostrará en la terminal un mensaje con la información acerca del error.

- Invocación correcta: “java -jar tinyRust+.jar entrada.rs”
- Invocación incorrecta: “java -jar tinyRust+.jar entrada.c”

Cualquier simulador de MIPS es válido para ejecutar el código ensamblador resultante de la ejecución del compilador, incluso el programa será compatible con piezas de hardware que implementen dicho set de instrucciones. A título personal de nosotros los desarrolladores, recomendamos la utilización de MARS (disponible para sistemas UNIX y Windows), así como también la utilización de QTSpim (disponible mediante flatpak para sistemas UNIX).

- MARS: https://courses.missouristate.edu/kenvollmar/mars/download.htm
- QTSpim: https://sourceforge.net/projects/spimsimulator/files/

Una vez cargado el ensamblador en alguno de estos simuladores con su posterior ejecución, se verá por el emulador de terminal de su simulador la salida del programa.