Licenciatura en Ciencias de la Computación

Compiladores
Manual de `tinyRU`

Ana Carolina Olivera
Universidad Nacional de Cuyo
Facultad de Ingeniería

Primer Semestre 2024

Índice general

1 Introducción

Durante el semestre se desarrollará el compilador para un lenguaje de programación reducido al que hemos llamado `tinyRU`. Aunque está basado en sus cadenas en `Rust` existen ciertos cambios en su estructura para poder desarrollar el 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 `tinyRU` está organizado en clases. `tinyRU` 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 `tinyRU`, también garantiza que no ocurran errores de tipo en tiempo de ejecución. `tinyRU` es fuertemente tipado.

Antes de empezar

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

```
java -jar tinyRU.jar archivo_entrada.ru [archivo_salida]
```

El compilador compila el archivo `archivo_entrada.ru`. 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.asm`. Tenga en cuenta que aunque la extensión es `.ru` no tiene exactamente la misma sintaxis que el lenguaje `rust` por lo que dará error si intenta compilar con un compilador de `rust`.
Los mensajes de error que el compilador genera cuando encuentra un error léxico, sintáctico o semántico no deben contener acentos para evitar problemas de incompatibilidades y no utilizar caracteres poco frecuentes. El compilador debe ser capaz de ejecutarse correctamente en cualquier sistema operativo actual.

2 Declaraciones

2.1. Clases: Structs

Todo el código de `tinyRU` está organizado en estructuras e implementaciones, juntas representan a las clases en el paradigma orientado a objetos. Por este motivo intercambiaremos las palabras estructura y clase para agilizar la lectura y comprensión. Múltiples estructuras pueden ser definidas en el mismo archivo. La definición de una estructura tiene la siguiente forma:

```
struct <IdStruct> [ : <IdStruct> ] {
 <Atributos>*
}
...
impl <IdStruct> {
 <Métodos>*
}
```

Todos los nombres de las estructuras son visibles globalmente (no hay clases privadas). Los nombres de estructuras deben **comenzar** con una **letra Mayúscula** y terminar con una **letra mayúscula** o una **letra minúscula**. Salvo el método que se invoca al crear una instanciación de una estructura o clase. Dicho método será identificado por tener como nombre un · únicamente. No puede haber dos *struct* con el mismo nombre en el mismo código fuente. No puede haber más de un *impl* para el mismo *struct* en el mismo código fuente.

2.2. El método start

Cada código fuente debe tener un método `start` que no tome parámetros formales. El método `start` debe estar definido al final de todas las struct. Un programa en `tinyRU` comienza ejecutando el método `start`.

2.3. Miembros

El cuerpo de una definición de struct consiste en una lista de definiciones de *atributos* para la clase definida por *struct*. En el cuerpo de *impl* para un *struct* estarán los métodos de dicho *struct*. Un miembro de una clase o struct es un *atributo* o un *método*. Un *atributo* de un *struct* `A` especifica una variable que es parte del estado de objetos del *struct* o 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 **público** salvo aquellos que tengan la palabra reservada `pri` del identificador de struct de los atributos que serán solo accesibles a la clase. Un *substruct* o 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 `st` serán métodos de clase. **No puede accederse a una variable de instancia en un contexto estático**.
Por ejemplo,

```
struct Mundo {
    pri Int a;
```

```
    Str b;
}

impl Mundo{
        .(){ a = 42;}
        fn get_a()-> Int { ret a; }
        st fn imprimo_algo()-> void {
                (IO.out_str("hola mundo"));
        }
}
...
struct Prueba{
...
Mundo c;

...
}
impl Prueba{

        .(){
                c = new Mundo();
                y = c.b;  // Acceso correcto al atributo de la clase
                z = c.a; //  Acceso incorrecto al atributo de la clase
                (c.imprimo_algo());
        }
}
start{
}
```

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 fragmento del archivo `lista.ru` que muestra un ejemplo del uso de métodos y atributos.

```
struct Cons Lista {
        Int xcar;
        Lista xcdr;

}

impl Cons{

        fn isNil() -> Bool{
        ret false;
        }

        .(Int hd, Lista xcdr){
        {
                xcar = hd;
                self.xcdr = xcdr;
        }
```

```
}
start{
}
```

En este ejemplo, la clase `Cons` tiene dos atributos `xcar` y `xcdr` y dos métodos `isNil` y el método $\cdot$. 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 la llamada a $c = new(1, l)$.
Esta notación es el pasaje de mensajes típico de la *orientación a objetos*. Puede haber definiciones de métodos $\cdot$ en muchas diferentes clases. El mensaje busca la clase del objeto `c` para decidir qué método $\cdot$ 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 de creación de instancia de clase o $\cdot$ 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 `tinyRU` para que los programadores desaloquen objetos. `tinyRU` tiene gestión automática de memoria. Los objetos que no van a ser utilizados por el programa son desacolcados por el recolector de basura (garbage collector) en tiempo de ejecución.

2.4. Herencia

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

```
struct 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 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 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 (struct) en algún lugar del código fuente. Debido a que `tinyRU` tiene una herencia única, se deduce que si se satisfacen ambas restricciones, el gráfico de herencia forma un árbol con `Object` como raíz.

2.5. Tipos

En `tinyRU`, 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

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

ejemplo, como parámetro formal en un método. Los tipos de todos los atributos deben estar también declarados.
La regla de tipo básica en `tinyRU` 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 (leer teoría).
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 `tinyRU` se inicializan por defecto para contener valores del tipo apropiado. Se cuenta además con el tipo especial `void`.

2.6. Definición de Métodos

La definición de un método tiene la forma:

```
[ st ] fn <idMétodo>([<parámetros 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 `st`. Si `st` 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. Dentro del método no puede haber identificadores con el mismo nombre de los parámetros formales del método. 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 que retorna un error:

```
struct A {
}
impl A { fn  f() -> Int { ret 1; }}
struct B : A {
}
impl B {fn f() -> Str { ret "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

Int a un *Str*. 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 · es llamada únicamente cuando un nuevo objeto es inicializado. La definición de · tiene la siguiente forma:

```
.(<IdStruct> <id>,...,<IdStruct> <id>) { <sentencias> };
```

2.7. Constructor

Una clase **siempre** debe tener un método ·.
· 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.

```
struct UnaSubclase: UnaSuperClase {}
impl UnaSubClase {
        .() {
        // La inicialización de la subclase se coloca aquí
    }
}
```

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

2.8. Declaración de atributos

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

```
[pri] <type> <id>;
```

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

```
[pri] Array <type> <id>;
```

Los atributos no pueden ser redefinidos.

2.9. Declaración de parámetros Formales

En cuanto a la declaración de los parámetros formales esta deberá 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 `tinyRU` son:

3.1. Expresiones aritméticas, de comparación y operadores lógicos

- `tinyRU` 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 `Int`. El tipo estático de la expresión es `Int`.

  Para el caso de ++ y – la sintaxis será `++<expr>` y `--<expr>`. Observar que la `<expr>` debe devolver un tipo `Int` y tiene precedencia más fuerte que los operadores unarios comunes (+ y -).

- `tinyRU` solo tiene división de enteros sin parte decimal.

- En `tinyRU` 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 `Int, 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).

3.2. self

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

```
self.id
```

Por ejemplo,

```
struct UnaClase {
    Str greeting;

}
impl UnaClase {
        .(Str greeting) {
        self.greeting = greeting;
        }
}
...
```

3.3. Llamadas a métodos

Las llamadas a un método en `tinyRU` suelen tener la siguiente forma:

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

Tenga en cuenta que en `tinyRU` 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.

3.4. New

Una expresión `new` tiene la forma `new <type>([<parametros actuales>])`. Para el caso de los arreglos la expresión new queda de la siguiente forma `new Int[<expr>]`.
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 · de la clase si es que existe.
El `new` para un objeto de tipo `Array` implica que se reserva el tamaño especificado por el valor de una expresión entera 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 Int a;

a = new Int[6];
```

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 porque es el valor por defecto de un entero.

3.5. Acceso a Atributos

En `tinyRU` 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 públicos salvo que se utilice la palabra `pri`. **Los atributos de superclases privados no son accesibles por su subclase** salvo a través de los métodos de la superclase.

4 Sentencias

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

4.1. Asignación

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

```
<LadoIzquierdo>=<Expresión>;
```

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

4.2. Sentencia Simple

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

```
(<Expresión>);
```

4.3. Bucle `while`

Un bucle `while` en `tinyRU` respeta la siguiente forma:

```
while (<condición>) <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`.

4.4. Bloques

Un bloque en `tinyRU` respeta la siguiente forma:

```
{ <sentencia>* }
```

Las sentencias se evalúan en orden de izquierda a derecha.
Los bloques en `tinyRU` 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). Esto quedará más claro al observar la gramática.

4.5. Condicionales

La condición tiene la forma:

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

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

4.6. Retorno de método

El retorno de un método tiene la forma:

```
ret <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.

5 Struct Base y Struct predefinidas

Ninguna de las clases o structs base poseen método · aunque las clases base deben inicializarse correctamente de acuerdo a su semántica. `tinyRU` cuenta con un pequeño conjunto de clases predefinidas y clases bases.

5.1. Struct predefinida Object

`Object`: La superclase o super estructura de todas las clases o estructuras de `tinyRU` (al estilo de la clase `java.lang.Object` de `Java`). En `tinyRU`, la clase `Object` no posee métodos ni atributos.

5.2. Struct Predefinida IO

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

- `st fn out_str(Str s)->void`: imprime el argumento.
- `st fn out_int(Int i)->void`: imprime el argumento.
- `st fn out_bool(Bool b)->void`: imprime el argumento.
- `st fn out_char(Char c)->void`: imprime el argumento.
- `st fn out_array_int(Array a)->void`: imprime cada elemento del arreglo de tipo `Int`.
- `st fn out_array_str(Array a)->void`: imprime cada elemento del arreglo de tipo `Str`.
- `st fn out_array_bool(Array a)->void`: imprime cada elemento del arreglo tipo `Bool`.
- `st fn out_array_char(Array a)->void`: imprime cada elemento del arreglo de tipo `Char`.
- `st fn in_str()->Str`: lee una cadena de la entrada estándar, sin incluir un carácter de nueva línea.
- `st fn in_int()->Int`: lee un entero de la entrada estándar.
- `st fn in_bool()->Bool`: lee un booleano de la entrada estándar.
- `st fn in_char()->Char`: lee un caracter de la entrada estándar.

Tanto `Object` como `IO` pueden ser utilizadas sin necesidad de ser heredadas. Para el caso particular de Array en caso de que el mismo no se encuentre inicializado la llamada al método deberá imprimir la leyenda "**El arreglo no se encuentra inicializado**".
`tinyRU` tiene otras cuatro clases básicas: (`Char, Int, Bool, Str`).

5.3. Array

La clase arreglo proporciona listas de tamaño estático de elementos de tipos primitivos (`Int`, `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()->Int`.

- `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 $0 \leq i \leq a.lenght() - 1$. Un arreglo no inicializado tiene tamaño 0. Es un error heredar o redefinir `Array`.
La creación de un arreglo se hace a través del new. En el momento de la creación los valores de los elementos del arreglo se inicializan a sus valores por defecto según su tipo primitivo.

- `new Array [Expresion]`.

5.4. Int

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

5.5. Str

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

- `fn length()->Int`. `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`.

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

5.6. Bool

La estructura `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`.

5.7. Char

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

`tinyRU` sólo 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 `tinyRU`. La estructura léxica se abarca en la Sección 6, la gramática generadora de código fuente en la Sección 7.

6 Lexemas y tokens en `tinyRU`

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

6.1. 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). Tenga presente que para este compilador se limitará el uso de caracteres raros o poco frecuentes en la lengua castellana para evitar problemas de compilación y testeo del compilador.

6.2. Comentarios

`tinyRU` tiene un tipo de comentario:

- `/?` Comentario simple: Todos los caracteres de desde `/?` hasta el final de la línea son ignorados. Se deben poder utilizar en los comentarios caracteres como #,$,%,&,@ , ¿, ¡, !, ?, y la ñ.

6.3. 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 por las clases que heradan de ella y podrán accederse en código fuera de la clase como atributo de una instancia de dicha clase, a no ser que estén declarados como `pri` en cuyo no serán visibles por nadie salvo la clase donde están definidos. 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 carácter de guión bajo o *underscore*.

Los identificadores de **tipo** (*struct*) comienzan con una **letra mayúscula** y terminan con una **letra mayúscula** o una **letra minúscula**.

Los identificadores de **objetos** comienzan con una **letra minúscula**. Los identificadores `self` y `void` son tratados especialmente por `tinyRU` al igual que la palabra `Array`. Los símbolos sintácticos especiales (por ejemplo, paréntesis, corchetes, operador de asignación, etc.) se verán directamente en la gramática.

6.4. Palabras clave

Las palabras claves de `tinyRU` son: `struct`, `impl`, `else`, `false`, `if`, `ret`, `while`, `true`, `nil`, `false`, `new`, `fn`, `st`, `pri`, `self`. En `tinyRU` las palabras clave son sensibles a mayúsculas y minúsculas.

6.5. Literales

Un literal es la representación en código fuente del valor de un tipo, como un número o un Str por ejemplo: 42 es un entero literal, `" hello world "` es un literal Str y `true` es un literal booleano.

6.5.1. Literales Enteros

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

6.5.2. Literales cadenas

Las cadenas o Str están encerrados entre doble comillas `"..."`. El tamaño máximo no debe superar los 1024 caracteres. Una cadena no puede contener EOF. Una cadena no puede contener el null (carácter `\0`). Cualquier otro caracter puede incluirse en una cadena. Para simplificar no se aceptarán caracteres raros o poco usados como ideogramas chinos, Hangeul, símbolos griegos, etc. Keep it Simple!

6.5.3. Literales caracteres

Los literales caracteres tendrán el siguiente formato (se agradece a Micaela del Longo y Federico Williamson por el aporte a la descripción de los literales caracteres).
`'\x'` donde `x` es cualquier caracter excepto 0 (cero). El valor del literal `\x` es `x`, a no ser que `\x` en conjunto conlleve un significado especial.
Por ejemplo, dado los siguientes literales de caracteres donde `\x` en conjunto conlleva un significado especial:

- En `'\t'` el valor del literal es el valor del caracter Tab.
- En `'\n'` el valor del literal es el valor del caracter salto de línea.
- En `'\r'` el valor del literal es el valor del caracter retorno de carro.

Por otro lado, en los siguientes literales de caracteres `\x` no conlleva un significado especial:

- En `'\a'` el valor del literal es el valor del caracter `a`, es decir, `'\a'` es equivalente a `'a'`.
- En `'\+'` el valor del literal es el valor del caracter `+`, es decir, `'\+'` es equivalente a `'+'`.
- En `'\''`, `'\\'` y `'\"'` el valor del literal es el valor del caracter `'`, `\` y `"` respectivamente.

Para simplificar no se aceptarán caracteres raros o poco usados como ideaogramas chinos, alfabeto Hangeul, símoblos griegos, etc. Keep It Simple!

6.5.4. Literal nulo

El literal nulo se representa mediante la palabra reservada `nil`. Por defecto las referencias en `tinyRU` 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 `tinyRU` .
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 mecanográfica aparecen los tokens devueltos por el Analizador Léxico.

⟨*program*⟩ ::= ⟨*Lista-Definiciones*⟩ ⟨*Start*⟩

⟨*Start*⟩ ::= `start` ⟨*Bloque-Método*⟩

⟨*Lista-Definiciones*⟩ ::= (⟨*Struct*⟩ | ⟨*Impl*⟩)*

⟨*Struct*⟩ ::= `struct idStruct` ⟨*Herencia*⟩? `{` ⟨*Atributo*⟩* `}`

⟨*Impl*⟩ ::= `impl idStruct {` ⟨*Miembro*⟩+ `}`

⟨*Herencia*⟩ ::= `:` ⟨*Tipo*⟩

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

⟨*Constructor*⟩ ::= `.` ⟨*Argumentos-Formales*⟩ ⟨*Bloque-Método*⟩

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

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

⟨*Visibilidad*⟩ ::= `pri`

⟨*Forma-Método*⟩ ::= `st`

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

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

⟨*Lista-Declaración-Variables*⟩::= `idMetAt`
| `idMetAt ,` ⟨*Lista-Declaración-Variables*⟩

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

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

⟨*Argumento-Formal*⟩ ::= ⟨*Tipo*⟩ `idMetAt`

⟨*Tipo-Método*⟩ ::= ⟨*Tipo*⟩ | `void`

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

⟨*Tipo-Primitivo*⟩ ::= `Str` | `Bool` |`Int` | `Char`

⟨*Tipo-Referencia*⟩ ::= `idStruct`

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

⟨*Sentencia*⟩ ::= `;`
| ⟨*Asignación*⟩`;`
| ⟨*Sentencia-Simple*⟩`;`
| `if (`⟨*Expresión*⟩`)` ⟨*Sentencia*⟩
| `if (`⟨*Expresión*⟩`)` ⟨*Sentencia*⟩ `else` ⟨*Sentencia*⟩
| `while (` ⟨*Expresión*⟩ `)` ⟨*Sentencia*⟩
| ⟨*Bloque*⟩
| `ret` ⟨*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-Método-Estático*⟩
| ⟨*Llamada-Constructor*⟩

⟨*ExpresionParentizada*⟩ ::= `(` ⟨*Expresion*⟩ `)` ⟨*Encadenado*⟩$^{?}$

⟨*AccesoSelf*⟩ ::= `self` ⟨*Encadenado*⟩$^{?}$

⟨*AccesoVar*⟩ ::= `id` ⟨*Encadenado*⟩$^{?}$ | `id [` ⟨*Expresión*⟩ `]`⟨*Encadenado*⟩$^{?}$

⟨*Llamada-Método*⟩ ::= `id` ⟨*Argumentos-Actuales*⟩ ⟨*Encadenado*⟩$^{?}$

⟨*Llamada-Método-Estático*⟩ ::= `idStruct .` ⟨*Llamada-Método*⟩ ⟨*Encadenado*⟩$^{?}$

⟨*Llamada-Constructor*⟩ ::= `new idStruct` ⟨*Argumentos-Actuales*⟩ ⟨*Encadenado*⟩$^{?}$
| `new` ⟨*Tipo-Primitivo*⟩ `[` ⟨*Expresion*⟩ `]`

⟨*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 [` ⟨*Expresion*⟩ `]` ⟨*Encadenado*⟩$^{?}$

7.1. Ejemplo

A continuación un ejemplo de código escrito en `tinyRU` .

```
struct Fibonacci {
        Int suma;
        Int i,j;

}
impl Fibonacci {
        fn sucesion_fib(Int n)-> Int{
                 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);
                }
                ret suma;
        }

        .(){
                i=0; /? inicializo i
                j=0; /? inicializo j
                suma=0; /? inicializo suma
                }
        fn imprimo_numero(Int num) -> void{
                (IO.out_str("f_"));
                (IO.out_int(num));
                (IO.out_str("="));
        }
        fn imprimo_sucesion(Int s) -> void{
                /?El valor es:
                (IO.out_int(s));
                (IO.out_str("\n"));
        }
}
start{
                Fibonacci fib;
                Int n;
                fib = new Fibonacci();
                n = IO.in_int();
                (IO.out_int(fib.sucesion_fib(n)));
}
```