Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

# Compiladores e Intérpretes

**Semántica de MiniJava- Documento Completo**[1]

Segundo Cuatrimestre de 2023

## 1 Introducción

Este documento una descripción (no exhaustiva) de la semántica del lenguaje de programación MiniJava. Como se ha visto a lo largo de la materia, MiniJava es en gran parte un subconjunto de Java y, por lo tanto, la mayoría de las sentencias en MiniJava tienen la misma semántica que en Java. La principal diferencia entre los lenguajes está en las sentencias y operadores que estos permiten. MiniJava elimina muchas características de Java tales como hilos, excepciones, variables de clase, métodos abstractos, arreglos, operaciones de punto flotante, etc.. Además, MiniJava restringe el uso de algunas características como la posibilidad de declarar, en una clase, varios métodos con el mismo nombre y diferente cantidad de parámetros. Este documento proporciona guías para determinar qué declaraciones de un programa MiniJava sintácticamente válido son semánticamente válidas (*i.e.*, compilan sin errores).

***Aclaración Importante:*** *En lo que sigue del documento siempre que se hable "clase" se hace referencia tanto a clases concretas como a interfaces. Cuando se refiera a una en particular se hablara específicamente de clase concreta o de interface. Además cuando se habla de unidad se refiere tanto a métodos como a constructores*

## 2 Consideraciones Generales

Cada declaración de una clase, método, variable de instancia o parámetro formal asocia a la entidad un nombre y determina algunas de sus características asociadas. Los programas MiniJava luego podrán hacer referencia a una entidad particular a través de su nombre. Por lo tanto, el compilador de MiniJava deberá generar y mantener un símbolo para cada entidad declarada. Estos símbolos deberán organizarse en tablas de símbolos, asociadas a clases y/o métodos, ya que diferentes entidades tendrán diferente alcance. Por ejemplo, un parámetro o una variable local de un método `met1` sólo podrá utilizarse dentro del cuerpo de `met1`, mientras que una variable de instancia en una clase `c1` podrá utilizarse en cualquier método no estático de `c1`. De esta manera, el objetivo de la tabla de símbolos es dar la posibilidad de obtener las características de una entidad a partir de su nombre en el contexto adecuado. Por ejemplo, si `a` es un parámetro de un método `met2`, cuando se hace referencia a `a` en el cuerpo de `met2` la tabla de símbolos deberá brindar todas las características de ese parámetro (*e.g.*, tipo, nombre, posición).

Una posible forma de organizar las tablas de símbolos es la siguiente. En primer lugar, todas las entidades de tipo clase se almacenan en la *tabla de símbolos de clase*. Luego, cada elemento de esa tabla deberá contener una (o varias) tabla(s) con métodos, constructores y variables de instancia. A su vez, cada método y cada constructor tendrá asociada una tabla propia para almacenar los parámetros y variables locales del método o constructor.

[1] Este documento, contiene todo lo presentado en Semantica de MiniJava- Parte I - Declaraciones

## 2.1 Declaraciones de clase concretas e interfaces

El nivel superior de un programa MiniJava es una secuencia de una o más declaraciones de clase. En la declaración de una clase concreta se declaran todas sus variables de instancia (atributos) y unidades (definiendo el cuerpo de las unidades declaradas). Mientras que en una declaración de interface se declaran los encabezados de los métodos que contien. Entonces, basicamente, las declaraciones de clase de MiniJava son como sus homónimas en Java, con la diferencia de que permiten menos combinaciones de elementos.

### 2.1.1 Herencia

Similar a como es en Java, en MiniJava es posible modelar herencia simple entre clases concretas o heredar (implementar) interfaces. La herencia tiene la misma semántica que en Java respecto a la visibilidad de los elementos herados del los ancestros. La principal diferencia con Java es que no se permite herencia multiple de y entre interfaces. Esto es una clase concreta solo puede heredar de otra clase concreta o de una interfaz, mientras que una interface como maximo puede heredar de una interfaz.

Al igual que en Java, la implementación de interfaces por una clase concreta, require que esta última tenga versiones concretas de todos los métodos que aparecen en esas interfaces.

En MiniJava toda clase concreta tiene una superclase, con excepción de la clase predefinida `Object`. Si se declara una clase concreta sin la palabra `extends`, entonces esa clase será por defecto una subclase directa de `Object`. Además, al igual que Java, MiniJava controla que cualquier tipo de herencia no sufra de **circulararidad**. Además al igual que en Java una clase concreta debe sobreescribir todos los métodos de las interfaces que implementa.

### 2.1.2 Clases Predefinidas

Al igual que en Java, en MiniJava hay clases concretas predefinidas que pueden utilizarse sin la necesidad de ser previamente declaradas y definidas. Se cuenta con dos clases predefinidas:

- **Object**: La superclase de todas las clases de MiniJava (al estilo de la clase `java.lang.Object` de Java). A diferencia de java, en MiniJava, la clase **Object** solo posee el siguiente método estático:
  - `static void debugPrint(int i)` que imprime un entero por la salida estandar y finaliza la linea.

  Con este método cualquier clase del sistema (dado que todas heredan de **Object**) tiene la capacidad de imprimir por pantalla un numero sin la necesidad de requerir de System
- **String**:La clase predefinida para caracterizar los objetos de los literales string. En MiniJava, la clase String no posee métodos ni atributos.
- **System**: Contiene métodos útiles para realizar entrada/salida (al estilo de la clase `java.lang.System` de Java). A diferencia de `java.lang.System` esta clase sólo brinda acceso a los streams de entrada (System.in) y salida (System.out), pero lo hace de manera oculta proveyendo directamente los siguientes métodos:
  - `static int read()`: lee el próximo byte del stream de entrada estándar (originalmente en la clase `java.io.InputStream`).
  - `static void printB(boolean b)`: imprime un `boolean` por salida estándar (originalmente en la clase `java.io.PrintStream`).
  - `static void printC(char c)`: imprime un `char` por salida estándar (originalmente en la clase `java.io.PrintStream`).

- `static void printI(int i)`: imprime un `int` por salida estándar (originalmente en la clase `java.io.PrintStream`).
- `static void printS(String s)`: imprime un `String` por salida estándar (originalmente en la clase `java.io.PrintStream`).
- `static void println()`: imprime un separador de línea por salida estándar, finalizando la línea actual (originalmente en la clase `java.io.PrintStream`).
- `static void printBln(boolean b)`: imprime un `boolean` por salida estándar y finaliza la línea actual (originalmente en la clase `java.io.PrintStream`).
- `static void printCln(char c)`: imprime un `char` por salida estándar y finaliza la línea actual (originalmente en la clase `java.io.PrintStream`).
- `static void printIln(int i)`: imprime un `int` por salida estándar y finaliza la línea actual (originalmente en la clase `java.io.PrintStream`).
- `static void printSln(String s)`: imprime un `String` por salida estándar y finaliza la línea actual (originalmente en la clase `java.io.PrintStream`).

Nótese que, a diferencia de Java, cada método de impresión tiene un nombre diferente, ya que en MiniJava una clase no puede tener más de un método con el mismo nombre.

### 2.1.3 Visibilidad de Declaración y Definición de Clases

Todas las clases declaradas son globalmente visibles. Aun así, no pueden declararse dos clases (concretas e interfaces) con el mismo nombre. De ocurrir esta situación se reportará un error semántico. Note que, de esta manera, no podrán declararse clases concretas o interfaces llamadas `Object`, `String` o `System`.

## 2.2 Declaración de Variables de Instancia (Atributos)

Sigue la semántica de Java para las variables de instancia (atributos) en clases concretas. En una clase concreta no puede haber más de una variable de instancia en con el mismo nombre. La visibilidad para las variables sigue la semántica de Java para los atributos que no tienen visibilidad explicita (los asume como public). Por otra parte, al diferencia de Java, no es posible declarar una variable con el mismo nombre que una definida en una superclase. Finalmente, si una variable de instancia es declarada de tipo clase, esa clase tiene estar declarada. Por último, no es posible declarar atributos de tipo void.

## 2.3 Declaración/Definición de Métodos

A diferencia de Java, en MiniJava no pueden declarase dos métodos con el mismo nombre dentro de una misma clase. La visibilidad para los métodos sigue la semántica de métodos *public* en Java. Además, al igual que en Java, si el método es declarado con un retorno de tipo clase, esa clase tiene que estar declarada.

Al igual que en Java, es posible declarar un método con el mismo nombre que un método definido en una superclase (en cuyo caso la subclase sobre-escribe el método de la superclase). Si una clase sobre-escribe un método, éste debe tener exactamente la misma cantidad y tipo de parámetros, el mismo tipo de resultado que el método sobre-escrito y el mismo tipo de método (estático o dinámico).

Finalmente, las interfaces no pueden contewner declaraciones de métodos static.

## 2.4 Constructores

Los constructores en MiniJava funcionan como en Java. La principal diferencia es que una clase concreta en MiniJava sólo puede tener declarado/definido un único constructor. Al igual que en Java, en caso que no se declare ninguno, el compilador le asignará un constructor por defecto (sin argumentos

y con cuerpo vacío) a todas las clases concretas (incluidas las predefinidas). Además si declara un constructor para una clase concreta debe tener el mismo nombre que la clase.

### 2.5 Declaración de Parámetros Formales

Los parámetros formales en las declaraciones de unidades siguen una semántica similar a la de Java. Un método no puede tener más de un parámetro con el mismo nombre. Finalmente, si un parámetro es declarado de tipo clase, esa clase tiene estar declarada.

## 3 Chequeos Semánticos

Los chequeos semánticos tienen dos objetivos. El primer objetivo es descartar programas inválidos, por ejemplo: un programa con una delcaracion `class A extends A;` debería ser rechazado porque tiene herencia circular. El segundo objetivo es recolectar información acerca de los tipos que será luego utilizada en la generación de código.

En las siguientes subsecciones se presentarán informalmente consideraciones necesarias para realizar el chequeo semántico de un programa MiniJava.

### 3.1 Chequeos Semánticos referentes a las Declaraciones

Básicamente, es necesario chequear que en toda declaración donde se utiliza un tipo no primitivo (ej: declaración de herencia, variables o métodos) el nombre haya sido declarado, es decir, que corresponda a una clase existente. Entonces, para determinar si el nombre utilizado es correcto, se consulta la tabla de símbolos de clases. Si la clase no está en esa tabla, esto quiere decir que no se ha declarado y por lo tanto hay un error semántico.

Existen otros chequeos que deben efectuarse sobre las declaraciones, algunos de los cuales ya fueron mencionados en secciones anteriores. Aquí se presenta una lista con mas ejemplos:

- No pueden declararse dos clases con el mismo nombre.
- Ninguna clase puede declarar dos métodos con el mismo nombre.
- Ninguna clase puede definir dos variables de instancia con el mismo nombre.
- Ninguna clase puede tener herencia circular (es decir, siguiendo la linea de ancestros extenderse a si misma).
- Ninguna clasec concreta puede tener mas de un constructor y si la clase no tiene ninguno se le genera uno predefinido sin parámetros y cuerpo vacío.
- Si un método `m1` es declarado en una clase `x` y tiene el mismo nombre que un método en la superclase, entonces el modificador de método (`static` o `dynamic`), el tipo de los argumentos y el tipo de retorno también deben coincidir.
- Una clase concreta debe sobre-escribir todos los métodos de las interfaces que implementa.
- No se puede definir una variable de instancia con el mismo nombre que de una varaible de instancia de un ancestro
- En una unidad, ningún parámetro puede tener el mismo nombre que otro parámetro.
- Las interfaces no pueden tener métodos `static`
- Alguna clase deber tener un método estático llamado `main`, el cual no posee párametros.

Estos controles suelen realizarse en la segunda pasada, como parte del *chequeo de declaraciones*, directamente sobre la tabla de símbolos. Es decir, para toda clase almacenada en la tabla de símbolos se controla si todos sus elementos fueron correctamente declarados. Aun así, es más simple realizar los controles relativos a los nombres repetidos mientras se crea la tabla de símbolos en el analizador sintáctico.

## 3.2 Resolviendo el uso de Nombres en el Cuerpo de Métodos y Constructores

Estos chequeos buscan controlar la existencia y recuperar la información de las entidades (si son variables, clases, métodos, etc.) de los nombres utilizados en las sentencias del cuerpo de las unidades. Estos chequeos suelen realizarse como parte del *chequeo de sentencias*.

Para determinar la correctitud semántica del uso de un nombre `x` en el cuerpo de un método (o constructor) `met` de una clase `C`, suponiendo que `x` es un **idMetVar** analicemos los siguientes casos:

1. `x` se usa para llamadar a un método y es el primer elemento de un encadenado (*e.g.*, `x(2).y.z(3)`) o no tiene encadenados (*e.g.*, `x(2)`). Entonces `x` debe ser un método visible en ese contexto de la clase `C`.

2. `x` se usa para llamada a un método y es una parte interna de un encandenado a la derecha de un punto (*e.g.*, `g().x(2)`). Entonces `x` debe ser un método de la clase del tipo del elemento a su izquierda en ese encadenado (en el ejemplo `x` debe ser un método del tipo de retorno declarado para `g()`).

3. `x` se usa para acceder a una variable y es el primer elemento de un encadenado (*e.g.*, `x.z(3)`) o no tiene encadenados (*e.g.*, `x`). Entonces `x` debe ser una variable local, un parámetro, o una atributo (en ese orden) visible en ese contexto de la clase `C`.

4. `x` se usa para acceder a un atributo y es una parte interna de un encandenado a la derecha de un punto (*e.g.* `y.x`). Entonces `x` debe ser una variable de instancia visible del tipo del elemento de la izquierda del encadenado (en el ejemplo `x` debería ser una atributo visible de la clase de `y`)

En cambio si `X` es un **idClase** tendremos los siguientes casos:

5. `X` aparece a la derecha de un `new`. Entonces `X` debe ser un constructor de la clase `X` (*e.g.*, `new x(3,2)`).

6. `X` se usa para acceder a los métodos estáticos de una clase (*e.g.*, `X.y(3,2)`). Entonces `X` debe ser el nombre una clase existente.

A continuación, para cada uno de estos casos, se explicará cómo resolver el nombre.

### Resolviendo los Nombres de Métodos (inciso 1)

Para resolver situaciones como las identificadas por el inciso (1) hay que buscar en la tabla de métodos de la clase que contiene la expresión donde se está usando el nombre. Si no se encuentra en esa tabla o no es visible resultará en un error semántico.

Note que la simpleza de este control es posible si se sigue la estrategia planteada para el *chequeo de declaraciones* (ver Sección 4.2.1), ya que para cada clase se agregarán todos los métodos heredados en su tabla de métodos. De no haber realizado esto, el control requeriría chequear las tablas de métodos de las clases ancestras hasta encontrar el nombre.

#### Resolviendo los Nombres de Métodos (inciso 2)

Para resolver los nombres como los del inciso (2) primero es necesario determinar la clase en la cual hay que buscar el método. Esta clase se determina por el tipo (estático) de la parte inmediatamente a izquierda del nombre en el encadenado. El tipo de la izquierda debe ser una clase (no un tipo primitivo o `void`), o de lo contrario se reportará un error semántico. Una vez identificada la clase de la izquierda se buscará el método de manera similar al caso del inciso (1) pero en esa clase.

#### Resolviendo Nombres de Variables y Parámetros (inciso 3)

Para resolver los nombres en este contexto es necesario, en primer lugar, saber si se hace referencia a un parámetro o variable local del método, estudiando la tabla de variables locales y parámetros asociada al método. Si no es un parámetro, entonces es necesario buscar en la tabla de atributos asociada a la clase en la cual se encuentra el método o constructor actual y ver si es un atributo visible. En caso de no serlo, ocurrirá un error semántico de nombre no definido.

#### Resolviendo Nombres de Atributos (inciso 4)

Para resolver los nombres como los del inciso (4) primero es necesario determinar la clase en la cual hay que buscar el atributo. Esta clase se determina por el tipo (estático) de la parte inmediata a izquierda del nombre en el encadenado. El tipo de la izquierda debe ser una clase (no un tipo primitivo o `void`), o de lo contrario se reportará un error semántico. Una vez identificada la clase de la izquierda se buscará el atributo en la tabla de atributos del tipo de la izquierda. Si no se encuentra en esa tabla resultará en un error semántico. Nuevamente la simpleza de este ultimo control es posible siguiendo la estrategia planteada para el *chequeo de declaraciones* (ver Sección 4.2.1), ya que para cada clase se agregarán todos los atributos heredados en su tabla de atributos.

#### Resolviendo Nombres de Constructores (inciso 5)

Para resolver los nombres en este contexto en primer lugar es necesario buscar en la tabla de símbolos de clase por la clase a la cual hace referencia el constructor. Si la clase no existe se reportará un error semántico de clase no declarada. En caso de existir, es necesario ver en su tabla de símbolos si existe el constructor al que se hace referencia.

#### Resolviendo Nombres de Clases (inciso 6)

Para resolver los nombres en este contexto simplemente es necesario ver que sea el nombre de una clase existente en la tabla de clases. Si la clase no existe se reportará un error semántico de clase no declarada.

### 3.3 Chequeos Semánticos de Expresiones y Sentencias

En esta sección se presentará informalmente la especificación del sistema de tipos de MiniJava y cómo esto influye al determinar las características necesarias para que una sentencia sea semánticamente correcta. Básicamente, la especificación determinará:

- qué tipos son válidos,
- qué expresiones son correctamente tipadas,
- cuál es el tipo de una expresión,
- qué sentencias son semánticamente correctas, y
- chequeos semánticos más allá de los chequeos de tipos.

Estos chequeos re realizan como parte del *chequeo de sentencias*.

### 3.3.1 Tipos Válidos

Desde el punto de vista semántico, en general, MiniJava cuenta con dos categorías de tipos: tipos primitivos (`int, char, boolean`) y tipos clase (incluyendo los predefinidos `Object, System, String`). En particular, los tipos clase se pueden dividir en concretos e interfaces. Por otra parte, para denotar los tipos clase, si `x` es un identificador válido de clase se dirá que $\mathcal{C}$(`x`) es un tipo. Otro caso particular, es el `null` que es un tipo clase especial que es compatible con cualquier tipo clase. Una última consideración especial es que `void` no termina de ser un tipo, sino, más bien, es la palabra reservada utilizada para denotar que un método no tiene un tipo de retorno.

#### Subtipos

Al igual que en Java, una relación importante entre los tipos de MiniJava es la de *subtipo*. Básicamente, si un tipo $T1$ es un subtipo de $T2$, entonces $T1$ puede representar todo valor de $T2$ y toda operación que funciona en $T2$ funcionará en $T1$. La relación de subtipo tiene las siguientes características:

- todos los tipos son subtipos de sí mismos (esto incluye a los tipos primitivos);
- para todo par de clases `X` e `Y`, si `X` tiene como ancestro a `Y` entonces $\mathcal{C}$(`X`) es un subtipo de $\mathcal{C}$(`Y`).

#### Conformidad

Dado que en MiniJava es posible definir subtipos y variables polimórficas, la igualdad como operador para chequear tipos no es suficiente. En este tipo de lenguajes se utiliza el concepto de *conformidad*. La conformidad establece que un tipo $T1$ conforma con un tipo $T2$ si y sólo si $T1$ es subtipo de $T2$ (recordar que un tipo siempre es subtipo de sí mismo). Como se verá más adelante, el concepto de conformidad será utilizado para chequear, por ejemplo, asiganciones, pasajes de parámetros, o expresiones de comparación.

### 3.3.2 Chequeando Expresiones

Uno de los objetivos del análisis semántico en MiniJava es asegurar estáticamente que las expresiones de un programa son correctamente tipadas. Para hacer este chequeo en una expresión $e$ será necesario chequear los tipos de cada subexpresión de $e$. Particularmente, será necesario llegar hasta las *hojas* de la expresión y propagar la información de tipos hacia arriba. Entonces, en cada nodo del AST correspondiente a un operador habrá que chequear que los tipos propagados por las subexpresiones sean compatibles con ese operador. A continuación repasaremos informalmente parte de las expresiones correctamente tipadas y sus respectivos tipos (lo que no se presente aquí se asume que se comporta como en Java).

#### Chequeando Literales

Todos los literales son en sí mismos correctamente tipados, y las reglas que los gobiernan son las siguientes:

- El tipo de un literal *entero* es `int`.
- El tipo de un literal *caracter* es `char`.
- El tipo de los literales `true` y `false` es `boolean`.
- El tipo de un literal *string* es $\mathcal{C}$(`String`).
- El tipo del literal `null` es un tipo especial que conforma con cualquier tipo clase.

### Chequeando Expresiones Unarias

Los operadores unarios `+` y `-` trabajan sólo con subexpresiones `int` y siempre retornan un resultado de tipo `int`. En caso de tener una subexpresión de otro tipo se producirá un error de tipos. Por otra parte, el operador unario `!` trabaja sólo con subexpresiones `boolean` y retorna resultados `boolean`. Al igual que con los demás operadores unarios, si la subexpresión es de otro tipo se deberá reportar un error de tipos.

### Chequeando Variables

El tipo del acceso directo un atributo o una variable local/parámetro utilizado en una expresión es determinado a partir de la tabla de símbolos como se mostró en la Sección 3.2. Si el identificador no se puede resolver utilizando las reglas de esa sección, entonces habrá un error de compilación.

### Chequeando Expresiones Binarias

Los operadores binarios matemáticos `+`, `-`, `*`, `/` y `%` sólo trabajan con subexpresiones de tipo `int` y devuelven resultados también de tipo `int`. Los operadores booleanos `&&` y `||` sólo trabajan con subexpresiones de tipo `boolean` y devuelven resultados de tipo `boolean`. Los operadores de igualdad `==` y `!=` trabajan con cualquier subexpresión de tipos conformantes (*i.e.*, relacionados a través de herencia) en cualquier dirección y siempre devuelve valores de tipo `boolean`. Los operadores relacionales `<`, `<=`, `>=` y `>`, sólo trabajan con subexpresiones de tipo `int` y devuelven resultado de tipo `boolean`.

### Chequeando `this`

El tipo de `this` es el mismo que el de la clase en que se está utilizando.

### Chequeando Paréntesis

El uso de paréntesis no cambia el tipo de una expresión. Por ejemplo: `((("hola")))` es una expresión de tipo $\mathcal{C}$(`String`).

### Chequeando llamadas a Métodos y Constructores

El tipo de una llamada a un método o constructor en una expresión es determinado a partir de la tabla de símbolos como se mostró en la Sección 3.2. Si el identificador no se puede resolver utilizando las reglas de esa sección, entonces habrá un error de compilación. Ademaás, para asegurar que las llamadas a métodos son correctamente tipadas, una llamada a un método cuya forma es `m(e1, ..., en)` debe satisfacer los siguientes requerimientos:

- El número de parámetros actuales debe coincidir con el número de parámetros formales.
- El tipo de la expresión correspondiente a cada parámetro actual debe conformar con el tipo asociado a cada parámetro formal.

El tipo resultante de una llamada a un método `m` es el tipo de retorno de `m`. Por ejemplo, considere los siguientes métodos:

```
int f(int x) { ... }
boolean g(int x, boolean y) { ... }
```

A continuación se presentan algunas expresiones que utilizan estos métodos, indicando cuál sería el resultado del chequeo de tipos:

| | |
|---|---|
| `f(2)` | es correctamente tipada y el tipo de la expresión es `int`. |
| `g(2)` | no es correctamente tipada porque el número de parámetros actuales es erróneo. |
| `g(2, 2)` | no es correctamente tipada porque el tipo de la expresión correspondiente al segundo parámetro actual no conforma con el tipo del parámetro formal. |
| `g(4, true)` | es correctamente tipada y el tipo de la expresión es `boolean`. |

Los constructores, a diferencia de los métodos, tienen como tipo de retorno el tipo de la clase del objeto construido y sólo se pueden utilizar en el contexto de un `new`.

#### Chequeando Expresiones Encadenadas Punto

Para chequear una expresión punto de la forma `e1.encadenado` primero es necesario chequear `e1`. La expresión `e1` debe ser correctamente tipada y el tipo de `e1` debe ser un tipo clase (no un tipo primitivo). En caso que se satisfagan estas condiciones se resolverá `encadenado` con las reglas vistas en la Sección 3.2.

#### Chequeando expresiones de asignación

La expresión de asignación $Izquierda$=$Derecha$ es correcta si y sólo si:

- $Derecha$ e $Izquierda$ son expresiones correctamente tipadas
- $Izquierda$ solo puede ser un Acceso[2] que:
  - si tiene encadenado, el ultimo elemento del encadenado debe ser una variable.
  - si no tiene encadenado, debe ser un acceso a variable
- El tipo de $Derecha$ conforma con el tipo de $Izquierda$.

### 3.3.3 Chequeando Sentencias

Las sentencias no tienen tipo por sí mismas. No obstante, para que una sentencia sea correcta todas las expresiones que utiliza deben ser correctamente tipadas y, en caso de ser necesario, tener los tipos adecuados.

#### Chequeando sentencias vacías

Este tipo de sentencias son inherentemente correctas y no es necesario realizar ningún chequeo semántico.

#### Chequeando una sentencia de llamada

Una sentencia de la forma $e$`;` será correcta si y solo si:

- $e$ es correctamente tipada
- $e$ tiene que ser un Acceso[2] que:
  - si tiene encadenado, el ultimo elemento del encadenado debe ser una llamada.
  - si no tiene encadenado, debe corresponderese con una llamada a metodo directa, llamada a método estatico o llamada a un constructor.

El tipo de $e$ es irrelevante

[2]Los Accesos son los operandos que permiten acceder a variables, hacer llamadas y que pueden tener encadenados. Para mas detalles ver las reglas de sintáxis de MiniJava

### Chequeando sentencias de asignación

La sentencia de asignación $e$ es correcta si y sólo si:

- $e$ es correctamente tipada
- $e$ es una expresión de asignación

### Chequeando sentencias de declaración de variables

La sentencia de declaración de variables locales es `var id =` $e1$ son correctas si y sólo si:

- `id` no es el nombre de un parámetro del método que contiene a la declaración o una variable local definida anteriormente en el mismo bloque o anteriormente en un bloque que contiene a al bloque que contiene la declaración.
- $e1$ es una expresión correctamente tipada y su tipo es T es distinto del tipo especial null

Una declaración de variable local correctamente tipada establecerá que:

- La variable local tendrá nombre `id` y su tipo será T
- Luego de la sentencia de declaración la variable será visible por todas las sentencias subsiguientes hasta que finalice el bloque actual (igual que en Java)

### Chequeando sentencias `if`

Una sentencia de la forma `if(`$e$`)` $S1$ {`else` $S2$} es correcta si y sólo si $e$ es una expresión correctamente tipada de tipo `boolean` y $S1$ {$S2$} es una sentencia correcta.

### Chequeando sentencias `while`

Una sentencia de la forma `while(`$e$`)` $S$ es correcta si y sólo si $e$ es una expresión correctamente tipada de tipo `boolean` y $S$ es una sentencia correcta.

### Chequeando sentencias `return`

Una sentencia de retorno de la forma `return;` es correcta si y sólo si el método que la contiene tiene como tipo de retorno `void`. Una sentencia de retorno de la forma `return` $e$`;` es correcta si y sólo si $e$ es una expresión correctamente tipada donde su tipo es $T$ y $T$ conforma con el tipo de retorno del método que contiene la sentencia `return`.

### Chequeando sentencias de bloque

Una sentencia de bloque es correcta si y sólo si todas sus sub-sentencias son correctas. Cuando se comienza el control de correctitud de un bloque es conveniente indicar que es el bloque actual de análisis.

### Chequeos correspondientes a sentencias dentro de métodos `static`

En el cuerpo de un método estático un acceso no puede comenzar con `this`, variables de instancia (atributos no `static`) o llamadas a métodos dinamicos. De encontrarse tal referencia deberá reportarse un error semántico.

# 4 Consideraciones de Implementación del Chequeador de Declaraciones de MiniJava

Como se mencionó anteriormente, para realizar adecuadmente el análisis semántico de los programas MiniJava será necesario realizar dos pasadas. A continuación se mencionarán algunas consideraciones que pueden ser útiles para el diseño de cada una de estas pasadas.

## 4.1 Primera Pasada

En la primera pasada, mientras el analizador sintáctico reconoce el archivo de entrada, irá construyendo la tabla de símbolos. Si se sigue la forma sugerida en la Sección 2, entonces a medida que el analizador sintáctico procesa declaraciones de clases irá agregando clases a la *tabla de símbolos de clase*. Cuando procese las declaraciones de miembros de una clase (métodos y atributos) los agregará a las tablas de métodos y atributos de esa clase. Además, en particular, cuando procese las declaraciones de un método agregará todos los parámetros a la tabla de parámetros de ese método. Siempre que se trate de agregar una entidad en la tabla de símbolos se chequeará si el nombre no esta repetido.

## 4.2 Segunda Pasada

Al terminar la primer pasada, si bien se generó por completo la tabla de símbolos, aún no se realizó ningún control semántico. En la segunda pasada se procederá a realizar dichos controles. Estos controles se denominan chequeo de declaraciones.

### 4.2.1 Chequeo de Declaraciones

En el chequeo de declaraciones se realiza directamente sobre la tabla de símbolos utilizando las entradas para las clases, atributos, métodos y constructores, para ver si estas fueron correctamente declaradas. En esta etapa no se chequea el cuerpo de los métodos, sino que sólo se chequea que sus encabezados (parámetros, modificadores y tipo de retorno) estén correctamente declarados. En la Sección 3.1 se presenta información un poco más detallada de los chequeos sobre las declaraciones.

Además, en esta etapa se controlará si algún conjunto de clases sufre de herencia circular. Es decir, para toda clase deberá chequearse que no sea ancestra de sí misma. En esta pasada también se actualizarán las tablas de métodos y las tablas de variables de las clases en base a la relación de herencia, proceso que denominamos *consolidación*. En particular, en la consolidación, se deberán agregar todos los métodos y las variables que la clase hereda de sus ancestros, con excepción de aquellos que esta sobre-escribe. Finalmente, cuando se vea la generación de código en la *etapa 5*, se generan los *offsets* asociados a las variables de instancia de cada clase.

### 4.2.2 Chequeo de Sentencias

En esta etapa se chequeará la correctitud del cuerpo de cada método que se haya declarado. Para esto es adecuado que anteriormente se haya realizado el *chequeo de declaraciones*, de manera que se pueda asegurar que todas las clases fueron correctamente declaradas y que sus tablas de métodos se actualizaron para incorporar los métodos que heredan.

El chequeo de sentencias se realizará sobre los AST. La idea general es diseñar los nodos del AST de manera tal que tengan el código capaz de realizar los chequeos semánticos adecuados. Por ejemplo, el nodo que representa un operador binario como el + debería tener asociado un método que sea capaz de detectar si puede operar correctamente con los tipos de los nodos correspondientes a sus sub-expresiones. Por lo tanto, si se sigue esta alternativa, todo el código para realizar el *chequeo de sentencias* quedará dentro de las clases utilizadas para diseñar el AST.

En este apunte se identificarán dos tipos de chequeo para esta etapa: el *chequeo de tipos* y las *resoluciones de nombres*. Los primeros identificarán si las expresiones usadas en las sentencias del cuerpo son correctamente tipadas, y son explicados con más detalle en la Sección 3.3. Las segundas,

corresponden a identificar a qué entidad se hace referencia cuando se utiliza un identificador en el cuerpo de un método/constructor, y son presentadas con más detalle en la Sección 3.2. Estos dos tipos de chequeos se realizan en conjunto, ya que el chequeo de tipos dependerá de que los nombres que aparecen en las expresiones y sentencias que se esten chequeando tienen que esta resueltos.

Un caso particular que requiere especial atención, es el de la sentencia de declaración de variables locales. Estas sentencias deben modificar dinámicamente las variables locales accesibles en un bloque. Los chequeos asociados a estos nodos deberían, además, considerar las restricciones asociadas a los nombres especificados en la Sección 3.3.

Otro aspecto que requiere una consideración distinguida son las asignaciones (expresiones y sentencias) y las sentencias de llamada. Estos elementos requieren que sus hijos en el AST tengan cierta estructura para ser validos. Para realizar los controles es usual aumentar la informacion tipo resultante de las expresiones. En particular se adjunta informacion al tipo indicando si al estructura de de la espresion resultante pueder destinataria de una asignacion o si termina en una llamada.