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

Compiladores e Intérpretes

Semántica de MiniJava

Segundo Cuatrimestre de 2013

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 variables locales. Este documento proporciona guías para determinar qué programas **MiniJava** sintácticamente válidos son también semánticamente válidos (*i.e.*, compilan sin errores).

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, la idea de la tabla de símbolos es posibilitar 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).

MiniJava permite que las expresiones y las declaraciones hagan referencias hacia adelante. Es decir, que se referencie a una entidad que sintácticamente se encuentra declarada más adelante en el archivo de entrada. Por lo tanto, un compilador de **MiniJava** debe ser de (al menos) "dos pasadas":

1. La primer pasada, realizada en conjunto con el análisis sintáctico del archivo de entrada, procesa todas las declaraciones construyendo la tabla de símbolos, y para cada método y constructor definido crea su correspondiente AST (árbol de sintaxis abstracta).

2. La segunda pasada, dada la tabla de símbolos y los ASTs, controla la correctitud semántica de todas las declaraciones y del código (mediante los AST) de todos los métodos y constructores definidos. Además, en esta pasada se realizará la generación de código.

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 tabla 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. Además, cada método o constructor tedrá su cuerpo representado mediante el AST asociado.

2.1. Declaraciones de Clase

El nivel superior de un programa **MiniJava** es una secuencia de una o más declaraciones de clase. En la declaración de una clase se declaran todas sus variables de instancia, métodos y constructores y se define el cuerpo de los métodos y constructor declarados. 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

Al igual que en Java, en **MiniJava** es posible modelar herencia simple entre clases. La herencia tiene la misma semántica que en Java cuando la clase padre tiene todos sus métodos (estáticos y dinámicos) y constructores públicos y sus variables de instancia privadas.

En **MiniJava** toda clase tiene una superclase, con excepción de la clase predefinida `Object`. Si se declara una clase sin la palabra `extends`, entonces esa clase será por defecto una subclase directa de `Object`. Además, al igual que Java, **MiniJava** controla que la herencia sea **no circular**.

2.1.2. Clases Predefinidas

Al igual que en Java, en **MiniJava** hay clases 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). En **MiniJava** la clase **Object** 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 con el mismo nombre. De ocurrir esta situación se reportará un error semántico. Note que, de esta manera, no podrán declararse clases llamadas `Object` o `System`.

2.2. Declaración de Variables de Instancia

Sigue la semántica de Java para las variables de instancia privadas no estáticas. En una clase no puede haber más de una variable de instancia en con el mismo nombre. Además, a diferencia de Java, en **MiniJava** el nombre de una variable de instancia debe diferir del nombre de la clase y del nombre de los métodos.

2.3. Declaración/Definición de Métodos

Siguen una semántica similar a los métodos públicos. A diferencia de Java, en **MiniJava** no pueden declarase dos métodos con el mismo nombre dentro de una misma clase. Un método tampoco puede tener el mismo nombre que la clase donde se encuentra declarado/definido. Por otra parte, 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 la misma cantidad y tipo de parámetros, el mismo tipo de resultado que el método sobre-escrito y el mismo modificador de método (`static` o `dynamic`).

Adicionalmente, al igual que en Java, en **MiniJava** los métodos estáticos no tienen objetos receptores, sino que son solicitados mediante el nombre de la clase que los posee (o hereda). Por lo tanto, el uso de nombres de variables de instancia o referencias a `this` en las expresiones del cuerpo de métodos estáticos producirán errores semánticos.

2.4. Declaración/Definición de Constructores

Los constructores en **MiniJava** funcionan como en Java. La principal diferencia es que una clase en **MiniJava** sólo puede tener declarado/definido un único constructor. Además, al igual que en Java, puede declararse una clase sin constructor y el compilador le asignará un constructor sin argumentos (con cuerpo vacío).

2.5. Declaración de Parámetros Formales

Los parámetros formales en las declaraciones de métodos y constructores siguen una semántica similar a la de Java. Un método/constructor no puede tener más de un parámetro con el mismo nombre, o con el nombre de una variable local al mismo.

2.6. Declaración de Variables Locales

Las variables locales tienen una semántica similar a la de Java. Un método/constructor no puede tener más de una variable local con el mismo nombre, o con el nombre de uno de sus parámetros.

3. Chequeos Semánticos

Los chequeos semánticos tienen dos objetivos. El primer objetivo es descartar programas inválidos, por ejemplo: un programa con una expresión `x = "hola"+ 5` debería ser rechazado porque tiene un error de tipos. El segundo objetivo es recolectar información acerca de los tipos que será luego utilizada en la generación de código.

Las declaraciones de entidades, así como también las sentencias y expresiones en el cuerpo de métodos/constructores pueden hacer referencia a nombres que identifican otros métodos, constructores, variables de instancia, parámetros o variables locales. Aun así, el control respectivo a la correctitud semántica del uso de un nombre debe realizarse en la segunda pasada. Esto se debe a la posible existencia de referencias hacia adelante por lo que, para hacer los controles con toda la información necesaria, se requiere de la tabla de símbolos y los AST producidos luego de procesar todas las declaraciones en la primer pasada (junto al análisis sintáctico).

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 completa:

- 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 definir variables de instancia con el mismo nombre que ella o que alguno de sus métodos.
- Si un método `m1` es declarado en una clase `x` y tiene el mismo nombre que en la superclase, entonces el modificador de método (`static` o `dynamic`), el tipo de los argumentos y el tipo de retorno también deben coincidir.
- En un método o constructor, ningún parámetro puede tener el mismo nombre que otro parámetro o que una variable local.
- En un método o constructor, ninguna variable local puede tener el mismo nombre que otra variable local o que un parámetro.
- 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 el tipo (si son variables, clases, métodos, etc.) de los nombres utilizados en las sentencias del cuerpo de los métodos/constructores. Estos chequeos suelen realizarse en la segunda pasada 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`, analicemos los siguientes casos:

1. `x` es sólo parte de una llamada a método, entonces `x` debe ser un método de la clase `C` (*e.g.*, `x(2)`).

2. `x` aparece a la derecha de una expresión punto y es parte de una llamada a método, entonces `x` debe ser un método de la clase del tipo de retorno del metodo de la izquierda (*e.g.*, `g().x(2)`).

3. `x` aparece a la izquierda de una expresión punto, entonces `x` debe ser una variable de instancia, una variable local, un parámetro, o una clase (*e.g.*, `x.z(3)`).

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

5. `x` no es parte de una expresión punto, ni de una llamada, entonces `x` debe ser una variable de instancia, una variable local, o un parámetro.

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 la clase que contiene la expresión donde se está usando el nombre. Si no se encuentra en esa tabla 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 izquierda de la expresión punto. 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/Parámetros/Clases (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 asociada a la clase en la cual se encuentra el método o constructor actual y ver si es una variable de instancia. Sino, es necesario ver si es el nombre de una clase en la tabla de símbolos de clase. En caso de no serlo, ocurrirá un error semántico de nombre no definido.

Resolviendo Nombres de Constructores (inciso 4)

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 Variables/Parámetros (inciso 5)

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 asociada a la clase en la cual se encuentra el método o constructor actual y ver si es una variable de instancia. En caso de no serlo, ocurrirá un error semántico de nombre no definido.

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 suelen realizarse en la segunda pasada como parte del *chequeo de sentencias*. Será tarea del compilador asegurar estáticamente que todas las definiciones de variables de instancia y métodos utilicen tipos válidos, y que las expresiones sean correctamente tipadas.

3.3.1. Tipos Válidos

En **MiniJava** se cuenta con dos categorías de tipos: tipos primitivos (`int, char, boolean, String, void`) y tipos clase. En particular, `void` no es un tipo sino que es la palabra reservada utilizada para denotar que un método no tiene un tipo de retorno. Por otra parte, para denotar los tipos clase, si `x` es un identificador válido de clase se dirá que $\mathcal{C}(\texttt{x})$ es un tipo.

Subtipos

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

- todos los tipos son subtipos de sí mismos;
- los tipos primitivos no tienen subtipos, ni son subtipos de ningún tipo (salvo de sí mismos);
- para todo par de clases `X` e `Y`, si `X` es una subclase de `Y` entonces $\mathcal{C}(\texttt{X})$ es un subtipo de $\mathcal{C}(\texttt{Y})$.

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. El conjunto de las expresiones correctamente tipadas y sus respectivos tipos se especifica mediante un sistema de reglas en las próximas secciones.

Chequeando Literales

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

- El tipo del literal `null` es $\mathcal{C}$(`Object`).
- El tipo de un literal *string* es `String`.
- 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`.

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 Identificadores

El tipo de una variable de instancia, una variable local o un método 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

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 parámetrosactuales 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 Punto

Para chequear una expresión punto de la forma `e1.llamada` 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 `llamada` con las reglas vistas en la Sección 3.2.

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 sentencias de asignación

La sentencia de asignación `id=`$e1$ es correctamente tipada si y sólo si:

- `id` es el nombre de una variable de instancia de la clase actual, o una variable local/parámetro del método actual; y
- $e1$ es una expresión correctamente tipada y su tipo conforma con el tipo de la variable/parámetro `id`.

Chequeando una sentencia simple

Una sentencia de la forma (e)`;`, donde e es una expresión, será correcta si y sólo si e es correctamente tipada. El tipo de e es irrelevante.

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 `for`

Una sentencia de la forma `for(id=`$e1$`;`$e2$`;`$e3$`)` S es correcta si y sólo si:

- `id=`$e1$ es una asignación correcta,
- $e2$ es una expresión correctamente tipada de tipo `boolean`,
- $e3$ es una expresión correctamente tipada de tipo que conforma con el tipo de `id`
- 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. Además, para el bloque asociado a un método con tipo de retorno distinto a `void`, el bloque debe asegurar que cada camino de ejecución posea una sentencia de retorno.

Chequeos correspondientes a sentencias en metodos `static`

En el cuerpo de un método estático no pueden aparecer referencias a variables de instancia o la palabra reservada `this`. De encontrarse tal referencia deberá reportarse un error semántico.

4. Consideraciones de Implementación para un Analizador Semántico 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 y los árboles sintácticos abstractos (ASTs). 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, constructores y atributos) los agregará a las tablas de métodos y atributos de esa clase. Además, en particular, cuando procese las declaraciones de un método/constructor agregará todos los parámetros y variables locales a la tabla de variables de ese método/cosntructor. Cuando procese el cuerpo de un método/constructor creará el AST del código y se asociará a la entrada de ese método/constructor.

Como se vío en la teoría de la materia, el AST es una forma arbórea de representar la estructura sintáctica de un programa. En particular, para el compilador de **MiniJava** sólo se utilizarán los AST para representar el código de métodos y constructores. De esta manera, a medida que el analizador sintáctico va reconociendo el cuerpo del método irá generando el AST que representa ese código. Por lo tanto, los nodos del AST deberán estar diseñados de manera tal que puedan representar los disntitos tipos de sentencias y los elementos que las componen. Por ejemplo, se debería tener un nodo que pueda representar un bloque, el cual tiene una lista de nodos para representar las sentencias dentro de ese bloque. Otro ejemplo es el nodo que representa un operador binario como el + detro de una expresión, el cual debería tener asociadas dos nodos correspondientes a expresiones (las que va a operar).

4.2. Segunda Pasada

Al terminar la primer pasada, si bien se generó por completo la tabla de símbolos (con los ASTs), aún no se realizó ningún control semántico. En la segunda pasada se procederá a realizar dichos controles. Estos controles se dividirán en dos etapadas: *chequeo de declaraciones* y *chequeo de sentencias*. Es conveniente realizar primero el *chequeo de declaraciones* así el *chequeo de sentencias* es más efectivo.

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, 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 (parámentros, tipo de retorno y variables locales) 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 controlorá 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 etapa también se actualizarán las tablas de métodos de las clases en base a la relación de herencia. En particular, se deberán agregar todos los métodos que la clase hereda de sus ancestros, con excepción de aquellos que esta sobre-escribe. Finalmente, cuando se vea la generación de código en esta etapa, se generán 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 y constructor 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: los *chequeos 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 3.2. 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 3.3. Estos dos tipos de chequeos se realizan en paralelo mientras se procesa el AST.

5. Chequeos y Acciones Opcionales

En esta sección se listan algunos chequeos y acciones adicionales que un compilador **MiniJava** puede realizar de manera opcional.

5.1. Reportar Múltiples Errores Semánticos

Dado que el analisis semántico de un programa **MiniJava** se realiza en una segunda pasada utilizando una estuctura enreiquecida, es posible reportar varios errores semánticos sin la necesidad de detener dicho análisis. A diferencia de lo ocurrido durante el análisis sintáctico, la recuperación de errores es mucho más sencilla, ya que el análisis semántico se realiza directamente sobre la tabla de símbolos y los AST en lugar de sobre el archivo fuente. Por una cuestión de simplicidad se recomienda **sólo** reportar 1 error semántico por sentencia.

5.2. Todos los caminos llevan al retorno

Al igual que en Java, en **MiniJava** es posible controlar estáticamente que cada camino de ejecución en el cuerpo de un método posea una sentencia de retorno. Esto sólo se realiza para los métodos cuyo tipo de retorno no sea `void`. Si algún camino no conduce a un `return` se debe reportar un error semántico.

5.3. Sentencias inalcanzables

Los compiladores de **MiniJava** pueden, al igual que los de Java, incorporar chequeos estáticos para detectar en cada método las porciones inalcanzables de código (*i.e.*, código en un bloque con posterioridad a una sentencia de retorno). Si hay código inalcanzable en algun método se debe reportar un error semántico.

5.4. Variables locales posiblemente no inicializadas

Al igual que en Java, en **MiniJava** es posible controlar estáticamente si al utilizar una variable local como parte de una expresión ésta posiblemente no ha sido inicializada.