### 3.6.1 Sintaxis Concreta y Abstracta

La sintaxis concreta (CS) de un lenguaje, es la interfaz con que interactúan los usuarios para crear programas.

La sintaxis abstracta (AS) de un lenguaje, es una estructura de datos que contiene la información principal del programa, pero sin los detalles contenidos en la sintaxis concreta. Las instancias de la AS se representan por árboles de sintaxis abstracta (AST).

Desde la perspectiva de los usuarios, la CS es la interfaz del lenguaje. Por su parte, la AS actua como una API para acceder a los programas por medio de herramientas de procesamiento (validadores, transformaciones y generación de código).

Existen dos estrategias para desarrollar un lenguaje por medio de la CS y la AS:

**Primero se define la CS**, la sintaxis abstracta se deriva, de la sintaxis concreta, ya sea de forma automática o por medio del uso de "*hints*" en la especificación de la sintaxis concreta.

**Primero se define la AS**, la sintaxis concreta es definida, en base a las especificaciones de la sintaxis abstracta. El meta-modelo de RAM ya existe y es basado en Ecore, por tanto TextRAM utilizará este enfoque.

A partir de la definición de la CS y la AS, se pueden crear programas mediante el uso de "parsers" o proyecciones.

Construcción de programas, por medio de *parsers*: en este enfoque, el AST es construído a partir de la CS. Un "*parser*" instancia y popula el AS, basado en la información textual del programa. (la definición del formato del CS es llamada *gramática*).

Construcción de programas, por medio de "**proyecciones**": en este enfoque, el AST es contruído en forma directa por las acciones del editor, y la sintaxis concreta se representa por medio de las reglas de proyección de AST.

#### 3.6.1.1 "Parsers"

En compiladores tradiconales e IDEs, los "parsers" se escriben manualmente, por medio de grandes programas monolíticos que leen un flujo de carácteres y utilizan recursión para crear la estructura del árbol (AST).

Por otro lado, los "parsers" se pueden generar automáticamente con la ayuda de una definición formal llamada gramática. Este es el enfoque que utilizan la mayoría de los compiladores actuales y los "languages workbenches".

La **gramática**, es una especificación de sintaxis escrita para un DSL (definición textual de la sintaxis concreta).

Al momento de elegir entre la generación manual o automática de los "parsers", se debe considerar lo siguiente: - El "parser" generado en forma automática, no ofrece el mismo rendimiento o reporte de errores en comparación con un "parser" hecho a la medida por un experto. + El "parser" generado en forma automática, disminnuye en gran medida el esfuerzo de construcción de un "parser", además el desarrolador que define el lenguaje no tiene que ser un experto en la tecnología que crea los "parsers".

#### 3.6.1.2 Gramática

La gramática, es la definición formal de una sintaxis concreta textual. Consiste en la producción de reglas que definen como debe verse una entrada válida de texto.

Resumen de la definición de la gramática según xtext documentation o el libro de xtend

#### 3.6.1.3 Formalismos del AST

Los formalismos del AST (también llamados meta-modelos), son utilizados para representar árboles o grafos. Por lo general, dichos formalismos son "meta-circulares" en el sentido que pueden describirse a sí mismos.

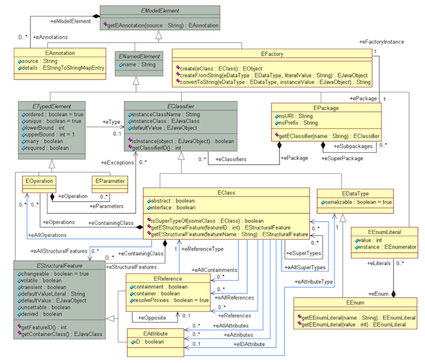
#### EMF Ecore

"Eclipse Modeling Framework" (EMF), es el corazón de las herramientas de modelado de Eclipse. Expone una amplia variedad de servicios y herramientas para persistir, editar y procesar modelos, también define reglas para la definición de la AS.

Su componente principal es *Ecore*, que actua como meta-modelo de los modelos EMF. Xtext usa Ecore, como base para el AS: a partir de la definición de la gramática. Xtext deriva el AS como una instancia de Ecore.

Los conceptos centrales de Ecore son: 1. *EClass*, representación de los elementos del AS o conceptos del lenguaje. 2. *EAtrribute*, representación de las propiedades primitivas de EClasses 3. *EReference*, representación de asociaciones entre EClases. Los EReferences opcionalmente pueden tener semántica de contenedores. 4. *EObject*, representación de instancias de EClasses (por ejemplo nodos AST). Cada EObject puede ser contenido de al menos una instancia de EReference.

Voelter, resume el modelo de Ecore en la Figura 1.



"Figura 1. Meta-modelo de Ecore visualizado como un UML"

### 3.6.2 Determinación de alcance y enlazamiento

La **determinación de alcance**, es el mecanismo principal detrás de la visibilidad y la resolución de referencias cruzadas. Desde el momento en que el DSL necesita estructurar el código, se necesita una implementación de la definición de alcance.

Por lo general, le determinación de alcance de una referencia cruzada, depende de:

1. El espacio de nombres en donde viven los elementos.
2. La ubicación dentro de la gran estructura del sitio que contiene la referencia cruzada.
3. Algún aspecto, que no necesariamente es estructural por naturaleza.

La determinación de alcance, también puede ser de ayuda para: 1. Popular el menu del "code-completion" del IDE cuando el usuario presiona Ctrl-Space en el sitio referenciado. 2. Validar una referencia existente.

El **enlazamiento** se refiere a la resolución de referencias basadas en en el nombre de las mismas. FALTA!!!!!!!!!

### 3.6.3 Restricciones

No todos los programas que están conformes con la estructura de un lenguaje (gramática, AS, meta-modelo), son válidos. Las definiciones del lenguaje incluyen restricciones que no pueden ser expresadas solamente por la estructura. Dichas restricciones adicionales son llamadas **restricciones**.

Las restricciones son condiciones "Boolean" que debe ser evaluadas como verdaderas, para poder indicar la validez de un modelo. Un mensaje de error debe ser reportado si la expresión a evaluar es falsa.

Se pueden distinguir dos grandes tipos de restricciones:

1. Restricciones que exigen que los elementos esten bien formados. Ejemplos: unicidad de los nombre en una lista de elementos, todas las variables deben estar definidas antes de utilizarlas.
2. Sistema de tipos: las reglas de los sistemas de tipos, son diferentes porque verifican la correcta definición de los tipos dentro de un programa. Por ejemplo, el sistema de tipos se asegura que no se pueda asignar un float a un int.

Las restrucciones pueden ser implementadas por cualquier lenguaje o "framework" que pueda consultar un modelo y reportar errores al usuario.

Para definir restricciones eficientes, se debe soportar las siguientes características:

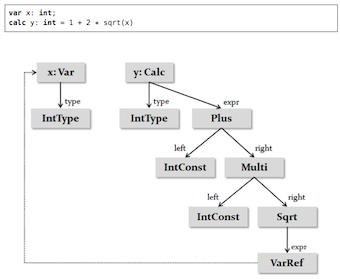
1. Debe permitir navegar y filtrar el modelo. Puede ser útil el soporte de "path expressions" (por ejemplo: aClass.operations.arguments.type como un mecanismo para buscar el tipo de los argumentos de todas las operaciones de una clase).
2. Se sugiere soportar "higher-order functions", para la escritura de algoritmos genéricos y estrategias transversales.
3. Es útil poder asociar una restricción en forma declarativa con los conceptos del lenguaje (o patrones estructurales) para aquellas instancias que necesiten ser ejecutadas.

### 3.6.4 Sistema de tipos

Los sistemas de tipos, implementan cálculo de tipos y verificaciones de tipos.

#### Conceptos básico del sistema de tipos

Para introducir los conceptos básicos del sistema de tipos, se hará uso del AST de la Figura 2.



"Figura 2. AST de un fragmento de código. Copiado de [12]"

El AST de la Figura 2, contiene cajas que representan instancias de los conceptos del lenguaje, las líneas sólidas representan contenedores y las líneas punteadas representan referencias-cruzadas.

Las obligaciones de un sistema de tipos son:

1. *Declarar tipos fijos"*: algunos elementos de los programas tienen tipos fijos. Los mismos no deben ser derivados o calculados. Por lo general permanecen igual y son definidos con anticipación. Algunos ejemplos son: la constante entera IntConst (cuyo tipo es IntType), el concepto de raíz cuadrada sqrt (cuyo tipo es double), así como las declaraciones de tipos por si mismas (el tipo de IntType es IntType, el tipo de DoubleType es Double-Type).
2. *Derivar tipos*: para algunos elementos del programa, el tipo debe ser derivado a partir de los tipos de otros elementos. Por ejemplo, el tipo VarRef (variable de referencia) es del tipo de la variable referenciada. El tipo de la variable es del tipo de su tipo declarado. Por ejemplo, el tipo de x y de la referencia a x es IntType.
3. *Calcular tipos comunes*: muchos de los sistemas de tipo, tienen alguna especie de jerarquía de tipos. En el ejemplo, IntType es un sub-tipo de DoubleType (por tanto, IntType puede ser usado, siempre que DoubleType sea esperado). El sistema de tipos debe soportar la especifcación de relación de subtipos.
4. *Verificaciones de tipo*: finalmente, un sistema de tipo debe verificar los errores de tipo y reportarlos al usuario. Para cumplir este objetivo, el lenguaje específica verificaciones del lenguaje, que son revisadas en tiempo de edición por el sistema de tipos, basado en los tipos calculados. En el ejemplo, un error de tipo puede ocurrir si algo con DoubleType fuera asignado a una variable IntType.