## Generación de tablas

Tal y como hemos mencionado, los algoritmos clásicos de minería de datos se basan en datos tabulares por lo que, para utilizar dichos algoritmos, necesitamos convertir los ASTs a tablas. Lo primero es definir los distintos tipos de subASTs que puedan tener una estructura homogénea, para poder almacenarlos en la misma tabla. Identificamos las siguientes 7 construcción sintácticas (tipos de subAST): programa (colección de sus tipos definidos), definición de tipo (clase, interfaz, enumerado o registro), definición de campo, definición de método, sentencia (o instrucción), expresión y tipo.

Las tablas 1-6 y 8 muestran toda la información almacenada para todas las construcciones sintácticas. A modo de ejemplo inicial, la Tabla 5 muestra la información almacenada para las sentencias. Además del nombre de la clase de la sentencia (su categoría sintáctica), guardamos las categorías sintácticas de sus tres primeros nodos hijos (si alguno de ellos no existe, se le asigna None) y la categoría sintáctica de su nodo padre. Como se comentó con anterioridad, también almacenamos el rol que la sentencia juega en el nodo padre. Asimismo, almacenamos la distancia desde el nodo raíz al actual (altura) y el número de arcos desde el actual al nodo hoja más distante (profundidad). El último valor almacenado para todos los nodos es el nivel de experiencia del programador.

Para realizar la traducción, tal y como se ha explicado anteriormente, hacemos uso del patrón de diseño *Visitor* [20]. Definimos siete caminos diferentes, uno para cada construcción sintáctica. Cada recorrido obtiene la información de una construcción sintáctica y la almacena en una tabla de la base de datos.

A continuación, explicamos las características extraídas para cada construcción sintáctica. Todas las tablas incluyen una característica común: el nivel del programador.

### Programas

La Tabla 1 muestra la información almacenada para cada programa. Incluimos diferentes porcentajes definidos en ese programa (clases, interfaces, enumerados…), todos ellos obtenidos de la sintaxis de sus hijos (construcciones de definición de tipos) e información sobre la implementación de las clases en paquetes.

Tabla 1: Características de programa.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Code in the default package | Si el programa implementa o no tipos en el paquete por defecto. | True or False. |
| Code in packages | Si el programa implementa tipos dentro de paquetes. | True or False. |
| Class percentage | Porcentaje de clases (de todos los tipos) definidas en ese programa. | [0, 100] |
| Interface percentage | Porcentaje de interfaces (de todos los tipos) definidos en ese programa. | [0, 100] |
| Enum percentage | Porcentaje de enumeraciones (de todos los tipos) definidas en ese programa. | [0, 100] |
| Number of types in packages | Número de tipos definidos en paquetes. | [0, n] |
| Number of types in the default package | Número de tipos definidos en el paquete por defecto. | [0, n] |
| Code in the default package | Si el programa implementa o no tipos en el paquete por defecto. | True or False. |

### Definiciones de tipo

En lo que respecta a las definiciones de tipo, calculamos distintas características. Algunas de ellas son obtenidas directamente de la API del compilador de Java (si el tipo es static o final), y otras son obtenidas a partir de la modificación del compilador (el número de métodos o de campos que son estáticos junto con sus porcentajes provienen de subárboles). La Tabla 2 muestra las características almacenadas para una definición de tipo.

Tabla 2: Características para la definición de tipos.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Category | Categoría sintáctica del nodo (nombre de la clase del nodo AST). | Class | Interface | Enumeration | Register. |
| Naming convention | La convención utilizada para el nombre del tipo. | Lower | Upper, Camel\_Low | Camel\_Up | Snake\_Case | Anonymous |
| Public Visibility | Si la visibilidad del tipo es pública o no. | True or False. |
| Static | Si el tipo es estático o no. | True or False. |
| Final | Si el tipo es final o no. | True or False. |
| Has Extends | Si el tipo extiende de otro tipo o no. | True or False. |
| Abstract | Define si es una clase abstracta o no. | True or False. |
| Inner Class | Define si es una clase anidada o no. | True or False. |
| Nested Class | Define si es una clase anidada estática o no. | True or False. |
| Strict TFP | Define si el tipo es StrictTFP o no. | True or False. |
| In default package | Define si la definición de tipo está o no en el paquete por defecto. | True or false. |
| Percentage of overloaded methods | Porcentaje de métodos sobrecargados declarados para ese tipo. | [0, 100] |
| Static Field Percentage | Porcentaje de campos que han sido declarados estáticos. | [0, 100] |
| Static Method Percentage | Porcentaje de métodos estáticos. | [0, 100] |
| Number of annotations | El número de anotaciones definidos para ese tipo. | [0, n] |
| Number of implements | El número de interfaces implementadas. | [0, n] |
| Number of generics | El número de genéricos definidos para ese tipo. | [0, n] |
| Number of methods | El número de métodos definidos en ese tipo. | [0, n] |
| Number of constructors | El número de constructores implementados en ese tipo. | [0, n] |
| Number of fields | El número de campos definidos para ese tipo. | [0, n] |
| Number of Static Blocks | El número de bloques estáticos definidos para ese tipo. | [0, n] |
| Number of Nested Types | El número total de tipos anidados definidos. | [0, n] |
| Number of Inner Types | El número de tipos anidados definidos. | [0, n] |
| Number of Static Nested Types | El número de tipos estáticos anidados definido. | [0, n] |

Almacenamos información sobre la categoría del nodo: clase, interfaz, enumeración o registro. Como ya hemos mencionado, el contexto obtenido de otras construcciones nos da nueva información para añadir nuevas características que podría ser de interés. Por ejemplo, el número de campos, de constructores o de métodos proviene de las definiciones utilizadas en esa definición de tipo. Otra información proviene de los modificadores de la definición de tipo o de los métodos ofrecidos por la interfaz ClassTree de la API de OpenJDK. También almacenamos el identificador del archivo, lo que nos permitirá crear más adelante modelos con características pertenecientes a varias construcciones.

### Definición de campo

Estas son las características definidas para las definiciones de campo (Tabla 3):

Tabla 3: Características para la definición de campos.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Visibility | La visibilidad del campo. | Public | Protected | Package | Private. |
| Parent Node | La categoría sintáctica del nodo padre. | Class | Interface | Enumeration | Register |
| Naming convention | La convención utilizada para el nombre del campo. | Lower | Upper | Camel\_Low | Camel\_Up | Snake\_Case |
| Type | El tipo del campo. | Type\* |
| Initial Value | El tipo del valor inicial, si existe. | Expression\* | None |
| Static | Define si el campo es estático o no. | True or False. |
| Final | Define si el campo es final o no. | True or False. |
| Volatile | Indica si el campo es volátil o no. | True or False. |
| Transient | Indica si el campo es transitorio o no. | True or False. |
| Number of annotations | El número de anotaciones declarado para ese método. | [0,n] |

\* Índica un dominio complejo. Este dominio se encuentra definido en el Anexo 9.1.

Cada vector de características obtenido de una definición de campo introduce información significativa sobre su tipo, junto con información de la aparición de diferentes palabras clave y, finalmente, información sobre el contexto.

Además de la categoría sintáctica, nuestro AST también introduce algunas diferencias significativas con el árbol del OpenJDK, definiendo por ejemplo nuevos roles hijos. Junto a este rol también incluimos información sobre la categoría sintáctica del nodo padre. El Anexo 9.1 presenta el dominio de cada característica sintáctica compleja para definiciones de campos y de métodos, sentencias, expresiones o tipos. Puede encontrar la referencia en las tablas como un superíndice en el nombre del dominio.

### Definición de método

Respecto a la definición de métodos, la Tabla 4 introduce aquellas características finales extraídas de esta construcción. La visibilidad de un método puede ser en nuestro caso: pública, privada, protegida o de paquete. Hemos creado categorías para representar el tipo de convención de los nombres de los métodos y las variables contenidas en el método (nos quedamos con aquella más presente). Diferenciamos entre las siguientes convenciones: lower, upper, camel\_low, camel\_up o snake\_case. Definimos también una nueva entidad para el dominio si la definición de un método es un constructor.

Además, priorizamos aspectos como la obtención de información más específica para características como el tipo de retorno. Entre sus posibles valores se encontrarían que pudiera ser este un tipo primitivo (Char, Int, Double, String, Byte, Long, Float, Short, Boolean, Array, Generic), una referencia, un array independientemente del número dimensiones o un tipo genérico, que introduce nuevos tipos como parámetros. El tipo de retorno también acepta void y constructor como posibles valores. También almacenamos información sobre los tres primeros parámetros del método (si alguno de ellos no existe, la característica se establece como None).

Tabla 4: Características para la definición de métodos.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Visibility | La visibilidad del método. | Public | Protected | Package | Private. |
| Naming convention | La convención usada para el nombre del método | Lower | Upper | Camel\_Low | Camel\_Up | Snake\_Case | Constructor |
| Main naming locals | La convención más frecuente en el nombrado de las variables. | Lower | Upper | Camel\_Low | Camel\_Up | Snake\_Case |
| Return Type | El tipo de retorno del método. | Type\*| Void | Constructor |
| First, second and third parameter | El tipo del parámetro correspondiente. | Type\* | None |
| Abstract | Define si el método es abstracto o no. | True or False. |
| Default Implementation | Define si el método utiliza la implementación por defecto. | True or False. |
| Static | Define si el método es estático o no. | True or False. |
| Final | Define si el método es final o no. | True or False. |
| Has Overrides | Define si el método se ha sobreescrito o no. | True or False. |
| Constructor | Define si el método es un constructor o no. | True or False. |
| StrictTFP | Define si el método es StrictTFP o no. | True or False. |
| Native | Define si el método es nativo o no. | True or False. |
| Synchronized | Define si el método está sincronizado o no. | True or False. |
| Number of parameters | El número de parámetros declarados. | [0, n] |
| Number of generics | El número de tipos genéricos declarados para ese método. | [0, n] |
| Number of throws | El número de excepciones declaradas en la claúsula *throw*. | [0, n] |
| Number of annotations | El número de anotaciones declaradas en el método | [0, n] |
| Number of statements | El número de sentencias declradas en ese método. | [0, n] |
| Number of local variables | El número de variables local declaradas. | [0, n] |
| Number of Inner Classes | El número de clases anidadas. | [0, n] |
| Number of Overloaded Methods | El número de métodos sobreescritos con la misma firma. | [0, n] |

\* Índica un dominio complejo. Este dominio se encuentra definido en el Anexo 9.1.

### Sentencias

Tabla 5: Características de sentencia.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Category | Categoría sintáctica del nodo (nombre de la clase del nodo AST). | Statement\* |
| First, second and third child | Categoría sintáctica del hijo correspondiente. | Statement\* | Expression\* | Type\* | None |
| Parent node | Categoría sintáctica del nodo padre. | Statement\* | Expression\* | MethodDefinition |
| Role | Rol del nodo actual en el nodo padre. | StatementRole\* |
| Height | Distancia en arcos desde el nodo actual hasta el nodo raíz. | [0, n] |
| Depth | Distancia en arcos desde el nodo actual hasta el nodo hoja más distante. | [0, n] |

\* Índica un dominio complejo. Este dominio se encuentra definido en el Anexo 9.1.

La Tabla 5 muestra la información almacenada para cada una de las sentencias. Además de guardar la categoría sintáctica de la sentencia para la que hemos establecido un dominio, almacenamos información sobre las categorías sintácticas de sus tres primeros nodos hijos (si alguno de ellos no existiera, por ejemplo, las sentencias unarias de incremento o decremento, el valor de la característica se establece como None) y la categoría sintáctica de su nodo padre.

Como ya hemos comentado anteriormente, también almacenamos el papel que desempeña esa sentencia en el nodo padre. Por ejemplo, si el padre es un bucle for, la sentencia podría desempeñar el papel de inicialización (primer nodo hijo) o el de actualización (tercer nodo hijo). En este caso, el papel de condición (segundo hijo) corresponde a un nodo de expresión. Por último, también almacenamos los valores de profundidad y altura del nodo en el AST.

Un cambio en el diseño original del AST es que hemos considerado que las interfaces definidas en las clases del *OpenJDK* BlockTree y ExpressionStatementTree no deben ser nodos del AST, ya que creemos que añaden ruido a los datos sin aportar información adicional. Por ejemplo, para el fragmento de código “if(){}”, consideraremos el contexto de la expresión entre paréntesis como el primer hijo. El segundo hijo será una lista con las instrucciones del bloque y el tercero es una lista con las instrucciones del else, si existe. Hacemos algo similar para otras sentencias, como for*,* while, do-while, etc.

### Expresiones

Tabla 6: Características de expression.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Category | Categoría sintáctica del nodo (nombre de la clase del nodo AST). | Expression\* |
| First, second and third child | Categoría sintáctica del hijo correspondiente. | Statement\* | Expression\* | Type\*| None |
| Parent node | Categoría sintáctica del nodo padre. | Statement\* | Expression\* | FieldDefinition |
| Role | Rol del nodo actual en el nodo padre. | ExpressionRole\* |
| Height | Distancia en arcos desde el nodo actual hasta el nodo raíz. | [0, n] |
| Depth | Distancia en arcos desde el nodo actual hasta el nodo hoja más distante. | [0, n] |

\* Índica un dominio complejo. Este dominio se encuentra definido en el Anexo 9.1.

La Tabla 6 muestra la información almacenada para las expresiones. En este caso, almacenamos una información similar a la que almacenamos para las declaraciones. De nuevo, además de la categoría sintáctica de la expresión, almacenamos las categorías sintácticas de sus tres primeros hijos y de su padre. También guardamos información sobre el papel que desempeña (rol), junto con los valores de la altura y la profundidad. Al igual que hicimos con BlockTree y ExpresionStatementTree, ya no consideramos *ParenthesizedTree*.

Como Java 8 introduce las expresiones lambda, nuestro sistema también es capaz de procesarlas, procesando de forma separada sus parámetros y su cuerpo como hijos de una expresión lambda. Nuestra nueva sintaxis crea un nuevo nodo categorizado como LambdaExpression, que siempre tiene dos hijos. Por un lado, el nodo LambdaExpressionParameters añadirá uno o más parámetros. Por otro lado, LambdaExpressionBody comprueba el bloque de código, que puede contener otras declaraciones, sentencias o expresiones, con o sin llaves.

Además de esto, como se comentó con anterioridad, las expresiones Unary, Binary o Literal introducen ahora una nueva sintaxis compleja, añadiendo información más explícita. La Tabla 7 muestra la abstracción de características seguida para estas categorías en nuestro enfoque.

Tabla 7: Abstracción de características para diferentes categorías sintácticas de expresión.

| **Símbolo** | **Clase del nodo AST** |
| --- | --- |
| ‘+’ | ‘-‘ | Arithmetic (Unary) |
| ‘++’ | PrefixIncrement (Unary) |
| ‘++’ | PostfixIncrement (Unary) |
| ‘- -' | PrefixDecrement (Unary) |
| ‘>>’ | Bitwise (Unary) |
| ‘!’ | Logical (Unary) |
| ‘+’ | ‘-‘ | ‘\*’ | ‘/’ | ‘%’ | Arithmetic (Binary) |
| ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘==’ | ‘!=’ | Comparison (Binary) |
| ‘&&’ | ‘||’ | Logical (Binary) |
| ‘&’ | ‘|’ | ‘^’ | Bitwise (Binary) |
| 5, “Hello world”, ‘a’, … | NullLiteral, IntLiteral, StringLiteral, DoubleLiteral, CharLiteral, LongLiteral, FloatLiteral, ShortLiteral, BooleanLiteral |

También hemos tratado de analizar la aportación de otras clases de la API como IdentifierTree, estudiando su necesidad y manteniendo sólo aquellos nodos que nos proporcionan lo que consideramos información significativa. Por ejemplo, no consideramos un único valor como un identificador. Además de esto, un nodo de anotación es considerado por nuestro sistema como un nodo hoja, a no ser que la anotación contenga argumentos.

### Tipos

Nuestro sistema introduce una nueva construcción sintáctica, denominada tipo. Nuestro sistema diferencia entre los tipos primitivos y las referencias, siendo éstas un nuevo tipo de nodo que aparece en lugares como en el tipo de retorno de método, o en una declaración de variable. A continuación, aparecen diferentes ejemplos que muestran lo que sería considera do como un tipo (texto en negrita):

public **int** myMethod() {…};

**float** myFloat = 5f;

**int[]** myArray = [1, 3];

**List<String>** list1 = new ArrayList<String>();

Tabla 8: Características de tipos.

| **Nombre** | **Descripción** | **Dominio** |
| --- | --- | --- |
| Category | Categoría sintáctica del nodo (nombre de la clase del nodo AST). | JavaType\* |
| Primitive | Sí el tipo es primitivo o no. | True or False. |
| Parent Node | Categoría sintáctica del nodo padre. | FieldDefinition | MethodDefinition | VariableDefinition | CastExpression | InstanceOfExpression |NewArrayExpression |
| Role | Rol del nodo actual en el nodo padre. | TypeRole\* |
| Height | Distancia en arcos desde el nodo actual hasta el nodo raíz. | [0, n] |
| Number of Generics | Distancia en arcos desde el nodo actual hasta el nodo hoja más distante. | [0, n] |
| Number of Dimensions | El número de dimensiones de la variable declarada. | [0, n] |

\* Índica un dominio complejo. Este dominio se encuentra definido en el Anexo 9.1.

La Tabla 8 muestra cómo representamos los tipos. Hemos introducido un nuevo conjunto de características similares a las de sentencia y expresión. Representamos información sobre la clase del tipo, información contextual sobre el padre, su rol o el número de genéricos o dimensiones que la variable que lo define ha declarado.

## Generación de los *datasets*

Las 7 tablas creadas describen información sobre construcciones sintácticas homogéneas, permitiéndonos obtener patrones comunes de expresiones o sentencias. No obstante, un programa posee múltiples subárboles de distintas construcciones, representado mediante información heterogénea (p. ej., un programa tiene una clase que define un método, donde se usa una sentencia que incluye varias expresiones distintas). Por ello, además de utilizar la información homogénea, debemos crear *datasets* que incluyan la información heterogénea y así poder minar patrones sintácticos heterogéneos y más expresivos.

La creación de los *datasets* heterogéneos se realiza mediante operaciones de disgregación de datos (*drill down*) que combinan la información de las distintas tablas [23]. Así creamos los 5 siguientes *datasets* heterogéneos:

1. Una instancia por cada tipo definido, incluyendo la información de su programa.
2. Una instancia por cada campo de un tipo definido, incluyendo la información del programa y el tipo del campo.
3. Una instancia por cada método de un tipo definido, incluyendo la información del programa y los tipos de retorno y de los tres primeros parámetros (None si no los hubiere).
4. Una instancia por cada sentencia, incluyendo la información del programa, tipo y método en el que se ha definido.
5. Una instancia por cada expresión, incluyendo la información del programa, tipo, método y sentencia en el que se ha definido.

Cada uno de estos modelos surge de la combinación de dos o más construcciones homogéneas. Así, conseguimos relacionar características de diferentes construcciones sintácticas como programas, definiciones de tipos y métodos. Estos nuevos *datasets* serán útiles para obtener patrones sintácticos formados por características de varias construcciones sintácticas. Al almacenar también para todas las instancias de los *datasets* homogéneos información sobre el nivel de experiencia del programador, podremos obtener de forma posterior patrones asociados a ese nivel.

Desde el punto de vista de su implementación, el tiempo de cálculo se ve aumentado según se incluyen nuevos conjuntos de datos para la creación de los *dataset* heterogéneos. Por ello, hemos optimizado esta operación en la medida de lo posible a través del uso de consultas SQL, que nos permiten obtener estos nuevos conjuntos de datos a partir de los conjuntos de datos homogéneos almacenados en la base de datos. Hemos utilizado las sentencias *where* y *join* para unir los datos de las diferentes tablas y hemos creado una tabla nueva en base de datos para guardar los identificadores de todos los nodos guardados y de sus padres.

En resumen, nuestro sistema crea 12 conjuntos de datos (*datasets*) para el minado de patrones sintácticos: 7 homogéneos y 5 heterogéneos.