Procesadores de lenguajes. Práctica 9.

ANÁLISIS SEMÁNTICO DE CABECERA

Índice

1.1 Código a utilizar

- Paquetes:
  - tinto: paquete que contiene TintoCompiler, clase principal de la aplicación.
  - tinto.ast: paquete que describe el árbol de sintaxis abstracta del compilador tinto.
  - tinto.parser: paquete que contiene el analizador léxico de Tinto, SintaxException, SemanticException y TintoHeaderParser.
  - tinto.parsejj: paquete que contiene la descripción de los analizadores semánticos utilizando la herramienta JavaCC. Contiene SemanticException y el fichero TintoHeaderParser.jj

1.2 El análisis en dos pasos del compilador de tinto

- El lenguaje tinto no utiliza ficheros de cabecera que permitan predeclarar el contenido de una librería. No existen ficheros “.h” o “.hpp”.
- No se puede compilar un fichero Tinto en una única pasada. En una única pasada sólo se conoce el contenido de la parte del fichero que ya se ha analizado, pero no se puede saber lo que queda por analizar. “Si en el código de una función A() aparece una llamada a B() desconocida, el compilador no sabe si trata de un error o si la función B() está definida en la parte del fichero que queda por analizar.
- La solución es que en el compilador de tinto se realizan dos pasadas. La primera con el analizador TintoHeaderParser. Que obtiene la declaración de las funciones definidas en el fichero analizado, pero sin analizar el cuerpo de la función.
- El resultado de esta primera pasada en una estructura de datos tinto.ast.struct.LibraryDeclaration que contiene la lista de funciones descritas en el fichero.

1.2 El análisis en dos pasos del compilador de tinto

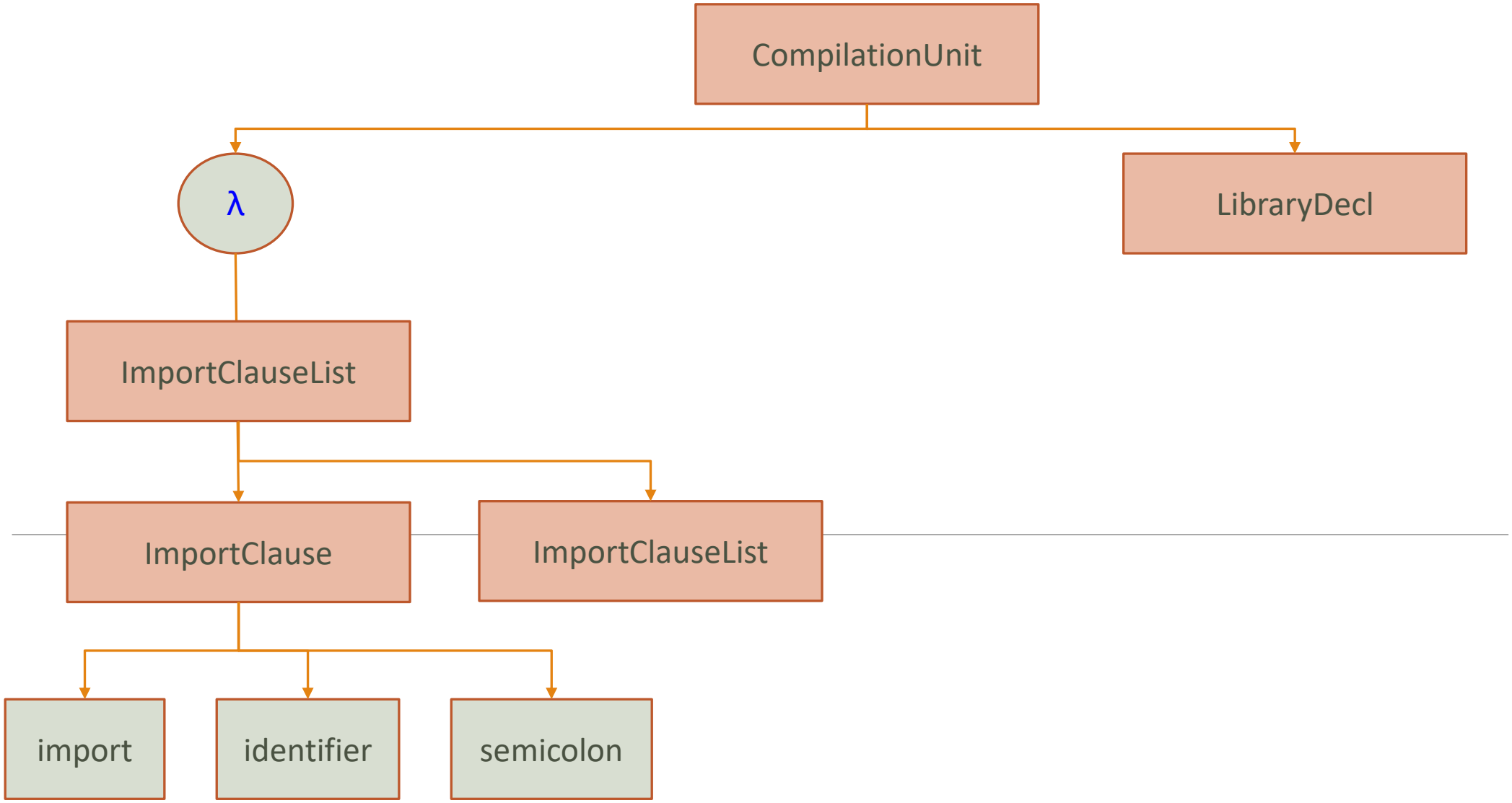
- Estas funciones son objetos tinto.ast.struct.Function en los que el campo body se deja vacío, no se incluye la lista de instrucciones de la función.
- La clase principal TintoCompiler, ejecuta el analizador TintoHeaderParser sobre el fichero Main.tinto y sobre todos los ficheros importados desde él, almacenando los objetos tinto.ast.struct.LibraryDeclaration obtenidos dentro de la tabla de símbolos tinto.ast.struct.SymbolTable.
- La segunda pasada se realiza con el analizador TintoParser. Se parte de la tabla de símbolos obtenida en la primera pasada y se vuelven a analizar los ficheros ".tinto" ya estudiados.
- Al comenzar el análisis de un fichero se declara el objeto LibraryDeclaration correspondiente como biblioteca activa (activeLibrary) dentro de la tabla de símbolos.
- Cada vez que se alcanza la definición de una función en el fichero, se busca el objeto tinto.ast.struct.Function. En la segunda pasada se analiza el cuerpo de la función y se almacena en body la lista de instrucciones.

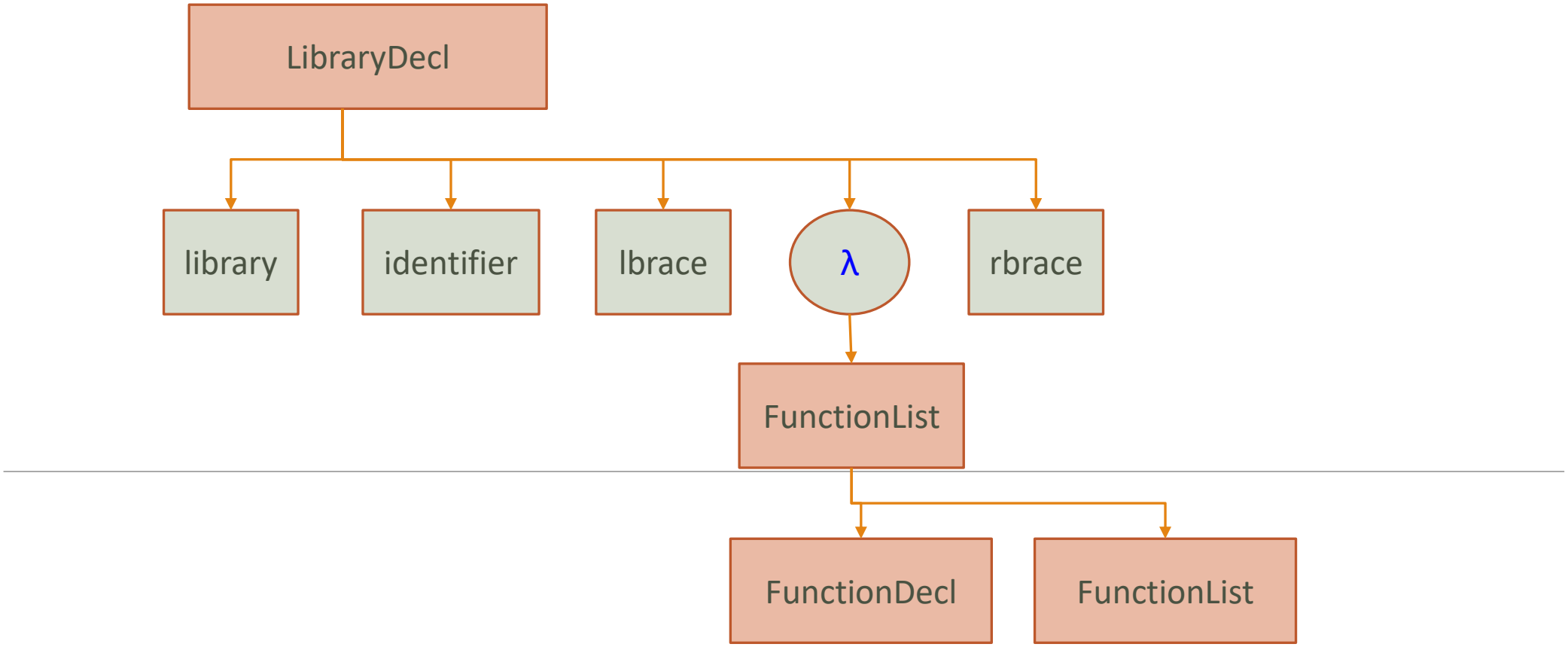
1.3 La clase tinto.parser.SemanticException

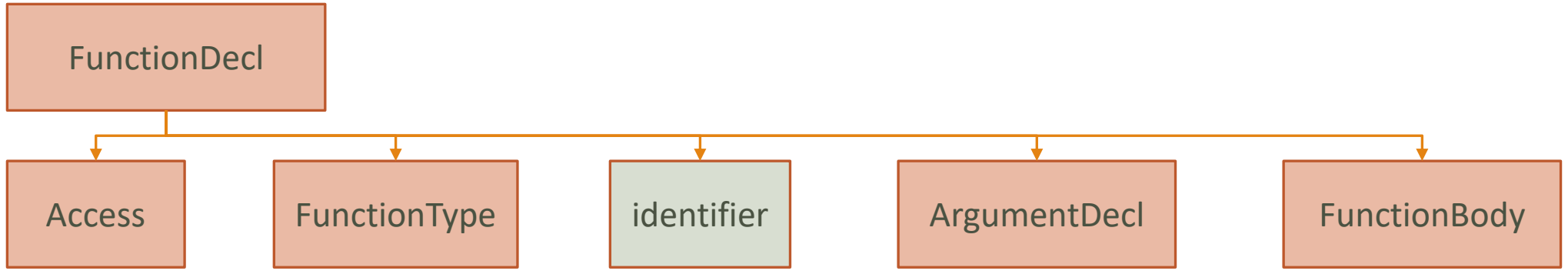
- Esta clase permite describir errores de tipo semántico. Se definen 13 errores semánticos diferentes.
- Se definen 13 constantes una por cada error, que representan el código de cada tipo de error.
- Hay un campo dónde se almacena el mensaje de error a mostrar.
- Este mensaje se construye a partir del código de error y de la información de la fila y columna del token de referencia del error.
- El método público toString() devuelve el mensaje de error almacenado.
- El método privado getExplanationForCode(int code) devuelve el mensaje de error asociado a cada código.

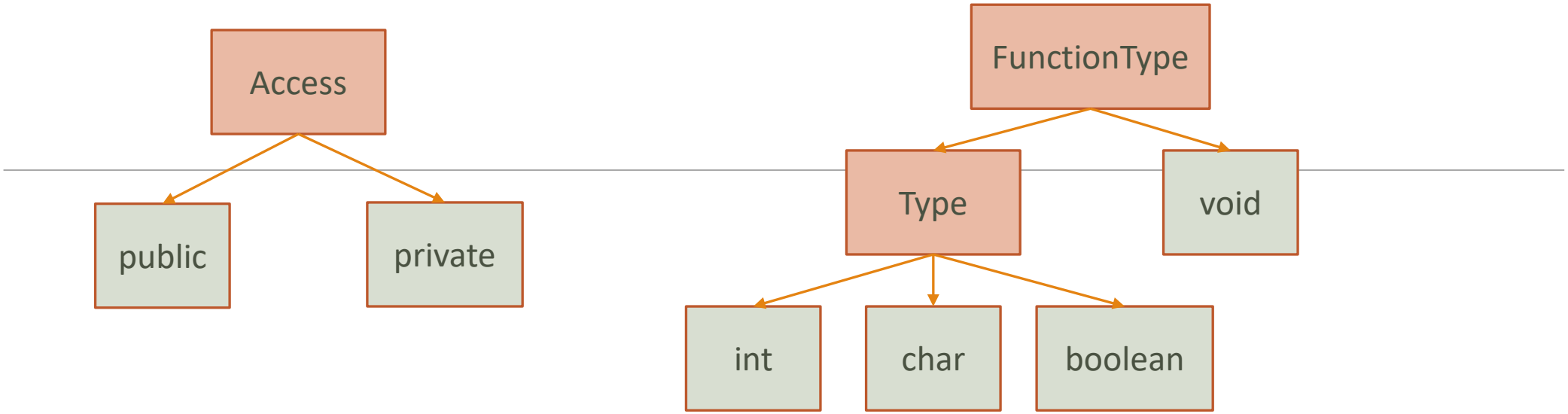
1.4 La clase tinto.parser.TintoHeaderParser

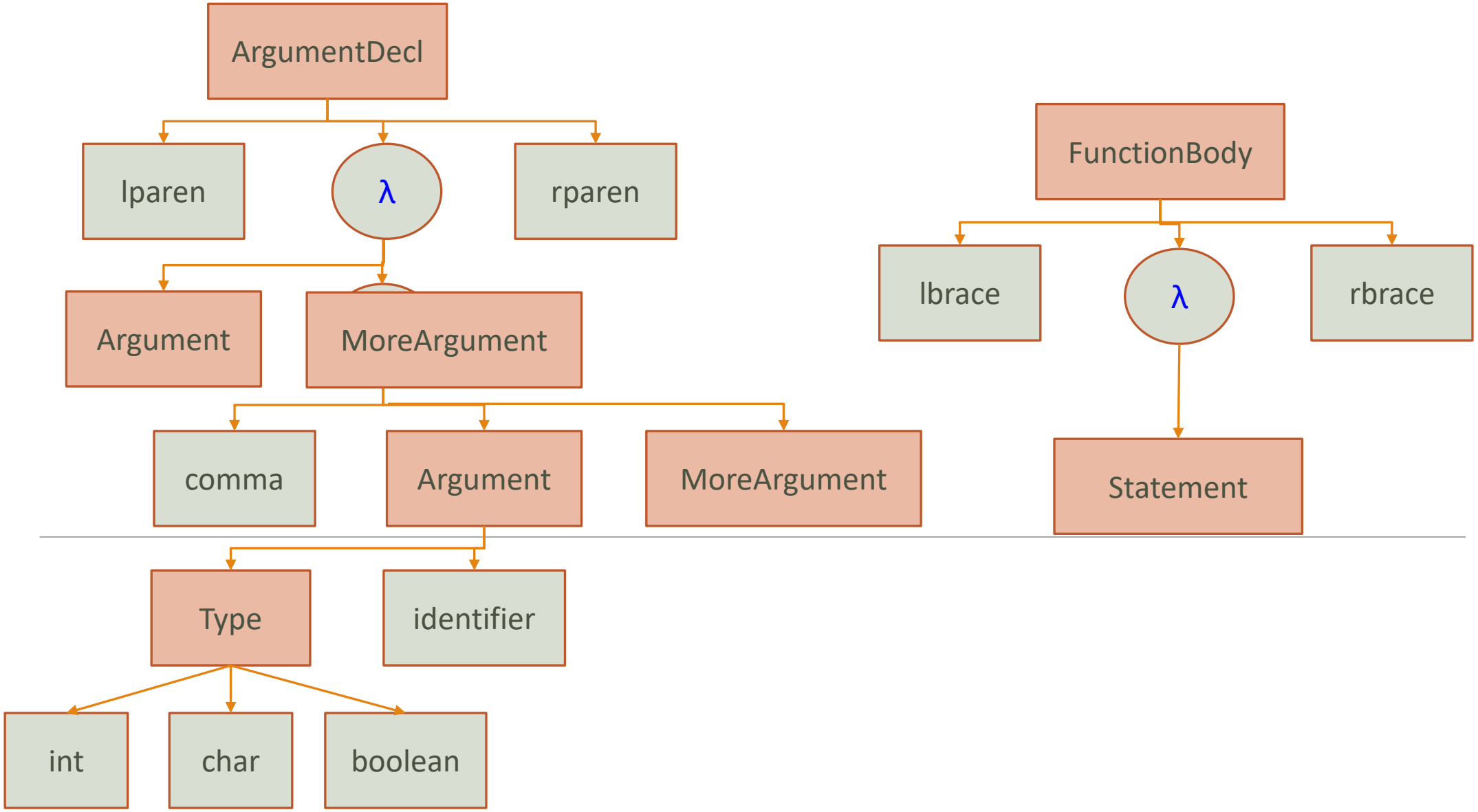
- Esta clase desarrolla un analizador sintáctico descendente recursivo similar al de la práctica 4, pero limitado a la información de la cabecera del lenguaje. Sustituimos FunctionBody por un método que lee un bloque de código entre llaves.
- El analizador desarrollado contiene técnicas de sincronización de errores similares a las presentadas en la práctica 6. El método skipTo() permite sincronizar el análisis a ciertos tokens.
- Para desarrollar el esquema de tratamiento de errores se han incluido tanto métodos del tipo parse…() que analizan sintácticamente un símbolo de la gramática, cómo métodos del tipo try..() que permiten sincronizarse en caso de error en algún método parse.


CompilationUnit
λ
LibraryDecl
ImportClauseList
ImportClause
ImportClauseList
import
identifier
semicolon


LibraryDecl
library
identifier
lbrace
λ
rbrace
FunctionList
FunctionDecl
FunctionList


FunctionDecl
Access
FunctionType
identifier
ArgumentDecl
FunctionBody


Access
public
private
FunctionType
Type
void
int
char
boolean


ArgumentDecl
lparen
λ
rparen
Argument
MoreArgument
comma
Argument
MoreArgument
Type
identifier
int
char
boolean
FunctionBody
lbrace
λ
rbrace
Statement

1.4 La clase tinto.parser.TintoHeaderParser

- No sólo desarrolla un análisis sentáctico sino también un análisis semántico. A medida que se realiza el análisis sintáctico se generan las estructuras del árbol de sintaxis abstracta que representa la biblioteca analizada.
- Es necesario incluir una serie de funciones que realizan verificaciones semánticas y acciones semánticas:
  - boolean verifyLibraryName(Token LibraryDeclaration): verificación semántica que comprueba que el nombre del fichero tinto corresponde con el nombre de la biblioteca.
  - boolean verifyNonDuplicateFunction(Token, Function, LibraryDeclaration): verificación semántica que comprueba que una cierta función no haya sido definida previamente. No se repiten funciones con los mismos nombres y los mismos tipos de datos.
  - boolean verifyNonDuplicatedArgument(Token, Function): verificación semántica que comprueba que en una función no se repiten argumentos con el mismo nombre.
  - void actionLibraryFunction(int, int, Token, Vector<Variable>,LibraryDeclaration): acción semántica que crea una función de una biblioteca.

1.4 La clase tinto.parser.TintoHeaderParser

- void actionAddArgument(int, Token, Vector<Variable>): acción semántica que añade un argumento a una lista de argumentos.

- Para realizar el análisis semántico es necesario modificar los métodos asociados a cada símbolo para añadir las verificaciones y acciones semánticas y para devolver los datos correspondientes:
  - LibraryDeclaration tryCompilationUnit(String): analiza el símbolo inicial. Añade el tratamiento de errores al análisis del símbolo CompilationUnit, devolviendo el mismo objeto que genera la función parse...() correspondiente.
  - LibraryDeclaration parseCompilationUnit(String): analiza el símbolo inicial. Devuelve como atributo sintetizado el objeto LibraryDeclaration que describe la biblioteca analizada. El argumento se utiliza para comprobar que el nombre de la biblioteca coincide con el nombre del fichero analizado.
  - void tryImportClauseList(Vector<String>): añade el tratamiento de errores al análisis del símbolo ImportClauseList. Este símbolo tiene como atributo heredado un vector de strings (imported) que permite almacenar los nombres incluidos en las claúsulas import.

1.4 La clase tinto.parser.TintoHeaderParser

- void parseImportClauseList(Vector<String>): analiza la lista de bibliotecas importadas. Utiliza ecomo atributo heredado un vector de strings (imported) que permite almacenar los nombres incluidos en las claúsulas import.
- void tryImportClause(Vector<String>): añade el tratamiento de errores al análisis del símbolo ImportClause. Utiliza un vector de strings como atributo heredado para almacenar el nombre de la biblioteca importada.
- void parseImportClause(Vector<String>): analiza una cláusula de importación. Utiliza el vector de strings como atributo heredado para añadirle la biblioteca importada.
- LibraryDeclaration tryTintoDecl(String, Vector<String>): añade el tratamiento de errores al análisis del símbolo TintoDecl. Tiene dos atributos heredados (el nombre de la biblioteca y un vector con los nombres de las bibliotecas importadas) y un atributo sintetizado (un objeto LibraryDeclaration con la descripción de la biblioteca analizada).
- LibraryDeclaration parseTintoDecl(String, Vector<String>): analiza el cuerpo de una biblioteca de Tinto, sea normal o nativa.

1.4 La clase tinto.parser.TintoHeaderParser

- LibraryDeclaration tryLibraryDecl(String, Vector<String>): analiza el cuerpo de la biblioteca incluyendo el tratamiento de errores. La función crea el objeto LibraryDeclaration con el nombre indicado y le añade la lista de biblitecas importadas. A continuación lanza el métod parse...() para analizar el símbolo LibraryDecl, pasándole como atributo heredado el objeto LibraryDeclaration.
- void parseLibraryDecl(LibraryDeclaration): analiza el cuerpo de la biblioteca. Utiliza el objeto LibraryDeclaration como atributo heredado. Verifica que el identificador analizado corresponde con el nombre de la biblioteca (que se extrajo del nombre del fichero ".tinto") y traslada el objeto como atributo heredado del símbolo FunctionList.
- void tryFunctionList(LibraryDeclaration): añade el tratamiento de errores al analisis del símbolo FunctionList. Utiliza como atributo heredado el objeto LibraryDeclaration que describe la biblioteca que se está analizando y lo pasa a la función parseFunctionList().
- void parseFunctionList(LibraryDeclaration): analiza la lista de funciones definidas en la biblioteca. Utiliza el objeto LibraryDeclaration para pasarlo como atributo heredado al símbolo FunctionDecl, que le añadirá la función que analice.

1.4 La clase tinto.parser.TintoHeaderParser

- void tryFunctionDecl(LibraryDeclaration): añade el tratamiento de errores al análisis del símbolo FunctionDecl.
- void parseFunctionDecl(LibraryDeclaration): analiza una función definida en la biblioteca y ejecuta la acción semántica actionLibraryFunction() para crear un objeto Function y añadirlo al objeto LibraryDeclaration (que recibe como atributo heredado).
- LibraryDeclaration tryNativeDecl(String, Vector<String>): analiza el cuerpo de la biblioteca nativa incluyendo el tratamiento de errores. El funcionamiento es similar al de las bibliotecas normales salvo la llamada al método setNative() del objeto LibraryDeclaration.
- void parseNativeDecl(LibraryDeclaration): analiza el cuerpo de la biblioteca nativa utilizando el objeto LibraryDeclaration como atributo heredado. Verifica que el identificador analizado corresponde con el nombre de la biblioteca (que se extrajo del nombre del fichero ".tinto") y traslada el objeto como atributo heredado del símbolo NativeFunctionList.
- void tryNativeFunctionList(LibraryDeclaration): añade el tratamiento de errores al analisis del símbolo NativeFunctionList.

1.4 La clase tinto.parser.TintoHeaderParser

- void parseNativeFunctionList(LibraryDeclaration): analiza la lista de funciones definidas en la biblioteca nativa.
- void tryNativeFunctionDecl(LibraryDeclaration): añade el tratamiento de errores al análisis del símbolo NativeFunctionDecl.
- void parseNativeFunctionDecl(LibraryDeclaration): analiza una función definida en la biblioteca nativa y ejecuta la acción semántica actionLibraryFunction() para crear un objeto Function y añadirlo al objeto LibraryDeclaration (que recibe como atributo heredado).
- int tryAccess(): añade el tratamiento de errores al anáisis del símbolo Access. Genera como atributo sintetizado el código correspondiente al tipo de acceso encontrado.
- int parseAccess(): analiza el símbolo Access. Genera como atributo sintetizado el código correspondiente al tipo de acceso encontrado: Access.PUBLIC_ACCESS o Access.PRIVATE_ACCESS.
- int tryFunctionType(): añade el tratamiento de errores al análisis del símbolo FunctionType. Devuelve como atributo sintetizado el código del tipo de dato de la función.

1.4 La clase tinto.parser.TintoHeaderParser

- int parseFunctionType(): analiza el tipo de dato de un método. Devuelve el código del tipo de dato como atributo sintetizado.
- int tryType(): añade el tratamiento de errores al análisis del símbolo Type. Devuelve como atributo sintetizado el código del tipo de dato.
- int parseType(): analiza un tipo de dato simple. Devuelve el código del tipo como atributo sintetizado.
- Vector<Variable> tryArgumentDecl(): añade el tratamiento de errores al análisis del símbolo ArgumentDecl. Genera como atributo sintetizado un vector de objetos Variable que describe los argumentos de la función analizada.
- Vector<Variable> parseArgumentDecl(): analiza los argumentos de un método. Genera como atributo sintetizado un vector de objetos Variable que describe los argumentos de la función analizada.
- void tryArgumentList(Vector<Variable>): añade el tratamiento de errores al análisis del símbolo ArgumentList. Utiliza como atributo heredado un vector de objetos Variable al que irá añadiendo los argumentos declarados en la función que se esté analizando.

1.4 La clase tinto.parser.TintoHeaderParser

- void parseArgumentList(Vector<Variable>): analiza la lista de argumentos de una función. Utiliza un vector de objetos Variable como atributo heredado. El proceso de análisis consistirá en añadir a este vector las declaraciones de los argumentos encontrados.
- void tryArgument(Vector<Variable>): añade el tratamiento de errores al análisis del símbolo Argument. Utiliza como atributo heredado un vector de objetos Variable al que añadirá la definición del argumento analizado.
- void parseArgument(Vector<Variable>): analiza un argumento de una función. Para ello recoge la información respecto al tipo de dato y el identificador del argumento y ejecuta la acción semántica actionAddArgument(). Esta acción se encarga de verificar que el identificador no está duplicado y de construir y añadir un objeto Variable con la descripción del argumento analizado.
- void tryMoreArguments(Vector<Variable>): añade el tratamiento de errores al análisis del símbolo MoreArguments. Utiliza como atributo heredado el vector de objetos Variable que almacena la información de los argumentos de la función analizada.

1.4 La clase tinto.parser.TintoHeaderParser

- void parseMoreArguments(Vector<Variable>) : analiza el resto de lista de argumentos de una función. Utiliza el vector de objetos Variable como atributo heredado con la lista de argumentos analizados.
- void parseFunctionBody(): analiza el cuerpo de una función. El analizador de cabecera no realiza ninguna acción semántica en este punto, así que la función se limita a recorrer un bloque de instrucciones, contando las llaves abiertas y cerradas para detectar el final del bloque de instrucciones.

1.5 Ejemplo JavaCC

```
 void parseFunctionDecl(LibraryDeclaration library)
:
  {
    int acc;
    int type;
    Token tid;
    Vector<Variable> args;
  }
  {
    acc = tryAccess()
    type = tryFunctionType()
    tid = <IDENTIFIER>
    args = tryArgumentDecl()
    FunctionBody()
    {
      actionLibraryFunction(acc,type,tid,args,library);
    }
}
```