IDE-Triangle
Guía de Implementación Rápida

Luís Leopoldo Pérez
Universidad Latina de Costa Rica
Facultad de Ingeniería en Sistemas de Información

luiperpe@ns.isi.ulatina.ac.cr

Septiembre – 2005

Resumen

IDE-Triangle es un entorno de desarrollo integrado diseñado para el lenguaje de programación didáctico Triangle, el cual es comúnmente utilizado en la enseñanza del diseño y construcción de compiladores. El presente documento pretende dar una descripción breve de la implementación del compilador en Java, el entorno de desarrollo y como conectarlo con dicha implementación.

1. ¿Qué es Triangle?

Triangle es un pequeño lenguaje de programación imperativo con estructura de bloques, desarrollado por el Profesor David Watt de la Universidad de Glasgow, ampliamente utilizado en docencia.

Existen varias implementaciones de compiladores para Triangle, siendo la versión en Java la más reciente. Dichas implementaciones son desarrolladas utilizando el descenso recursivo como técnica de compilación, haciendo que el código sea fácil de entender y modificar, lo cual lo hace apto para la enseñanza del diseño y la construcción de compiladores.

Los programas en Triangle son convertidos a un código objeto, el cual es interpretado por una máquina virtual llamada Máquina Abstracta de Triangle (TAM por sus siglas en inglés).

1.1. Ejemplo de programa en Triangle

El siguiente programa en Triangle calcula la serie de Fibonacci para una cantidad determinada de iteraciones.

```
! Calculates the fibonnaci sequence.
! Luis Leopoldo Pérez, Sep. 2005

let
   var i: Integer;
   var j: Integer;
   var k: Integer;
   var num: Integer
in
begin
  getint (var num); ! Number of iterations.
  i:= 1;
  j:= 2;
  while (num > 0) do
  begin
      putint(i);
      put(' ');
      k:= i+j;
      i:= j;
      j:= k;
      num:= num - 1;
  end;
end
```

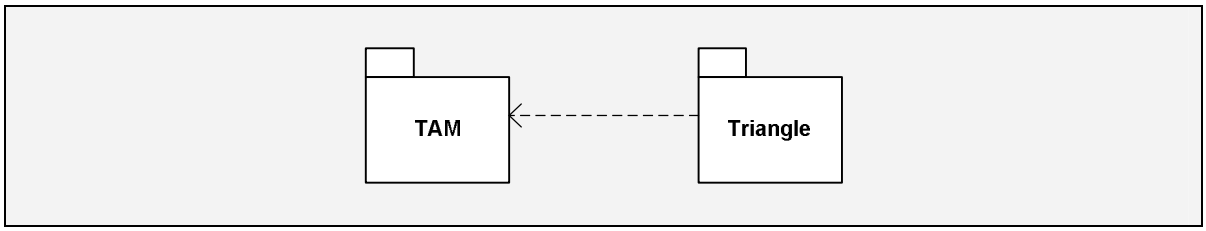
1.2. Implementación del compilador en Java

Una de las más recientes implementaciones del compilador de Triangle fue desarrollada por David Watt y Derryck F. Brown para su libro *Programming Language Processors in Java*.

El programa en Java se encuentra subdividido en varios paquetes, dentro de los cuales se encuentran las distintas clases e interfaces para las fases del proceso de compilación.

Hay dos paquetes principales: Triangle y TAM.

En el paquete TAM se encuentran todas las clases que definen la máquina virtual, el intérprete y el desensamblador de programas objeto. En el paquete Triangle está todo el compilador como tal.



El paquete Triangle se encuentra subdividido en otra serie de paquetes, representando esta cada fase del compilador. Los paquetes contenidos dentro de Triangle son:

- AbstractSyntaxTrees: contiene las clases que definen todas las estructuras de árboles de sintaxis para los procesos de análisis sintáctico, contextual y generación de código. Se puede considerar como el paquete más importante del compilador ya que casi todo el proceso depende de la interfaz Visitor, la cual se explicará más adelante.

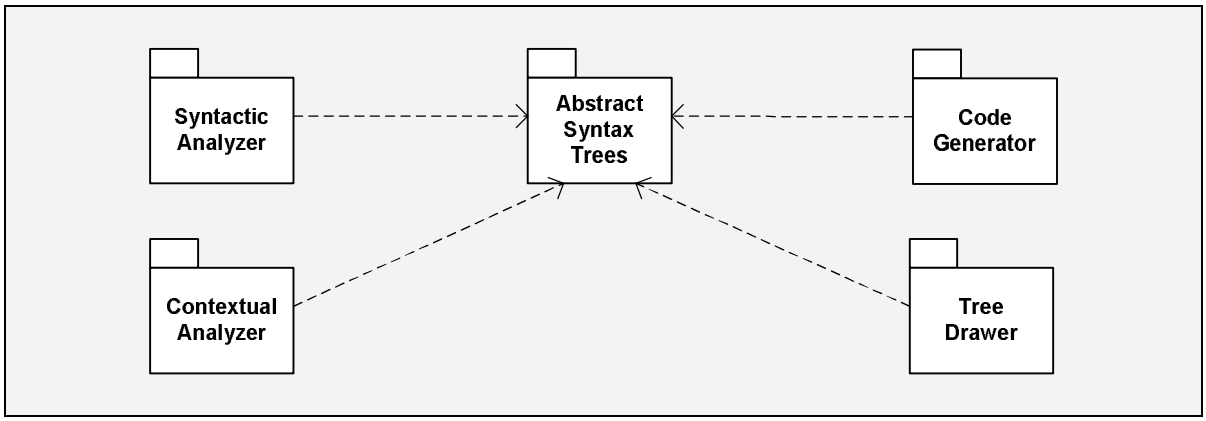
- SyntacticAnalyzer: contiene las clases que realizan el análisis léxico y sintáctico, generando los árboles de sintaxis abstracta que luego utilizarán el analizador de contexto y el generador de código.

- ContextualAnalyzer: contiene las clases que realizan el análisis de contexto, haciendo las revisiones de tipo y alcance pertinentes. Depende de la interfaz Visitor, la cual al implementarla le permite recorrer el árbol de sintaxis.

- CodeGenerator: contiene las clases que realizan la generación de código para la TAM. Al igual que el analizador de contexto, depende de la interfaz Visitor para poder recorrer el árbol de sintaxis.

- TreeDrawer: es un paquete experimental, dentro del cual se encuentran algunas clases para graficar los árboles de sintaxis. Aún no se encuentra terminado.

El siguiente diagrama muestra los distintos paquetes y su interdependencia:



1.2.1. La superclase AST

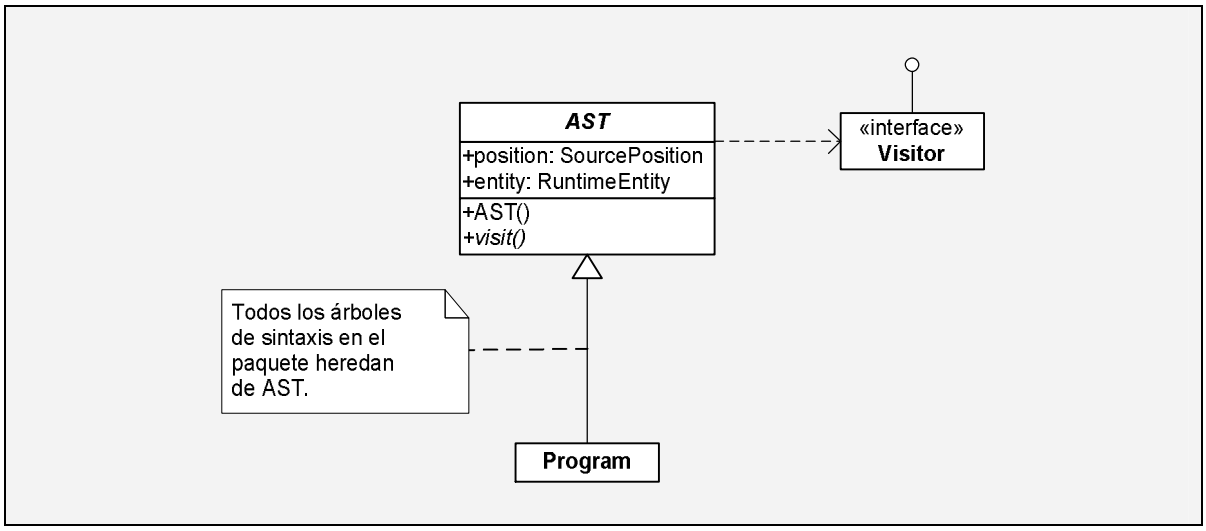
Dentro del paquete AbstractSyntaxTrees, se encuentra la superclase AST, la cual define el comportamiento mínimo que cada clase para un árbol de sintaxis debe tener. Básicamente obliga a que sus extensiones tengan:

- La ubicación del código dentro de la fuente para poder reportar errores, representado por la clase SourcePosition. Además, un método que permita obtener dicha ubicación llamado getPosition().

- Una entidad de tiempo de ejecución para efectos de generación de código, llamada RuntimeEntity.

- El método visit(), el cual permite el recorrido del árbol y utiliza la interfaz Visitor.

- Un constructor por *default.*



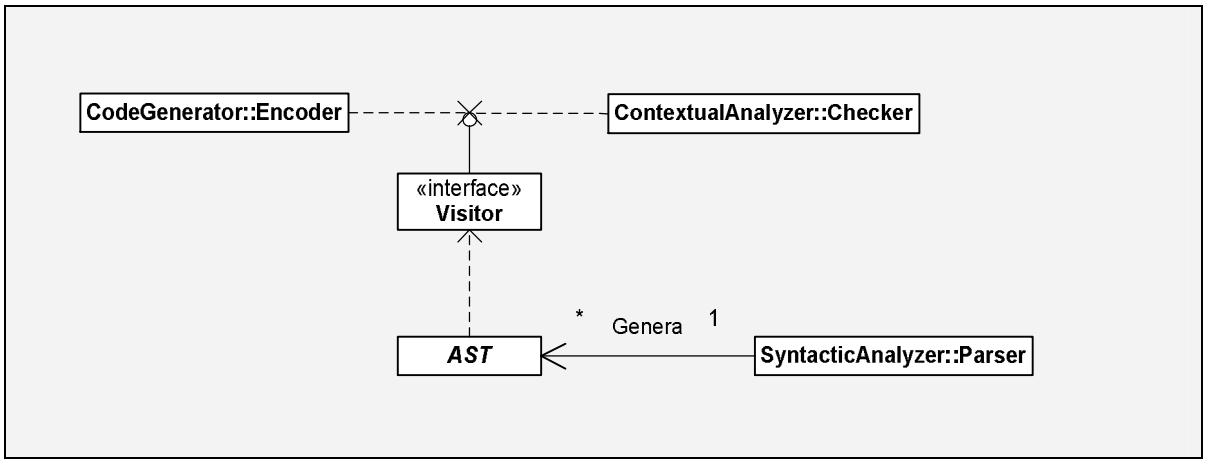
El diagrama anterior muestra la clase abstracta AST y sus relaciones.

1.2.2. La interfaz Visitor

La interfaz Visitor es uno de los elementos más importantes del compilador, ya que es la única manera para recorrer un árbol de sintaxis abstracta completo. Esta interfaz lo único que contiene son los métodos para visitar cada tipo de árbol de sintaxis, sin "cuerpo" alguno.

La importancia de esta interfaz radica en que el analizador contextual y el generador de código la implementan, ya que deben hacer un recorrido completo de todos los árboles de sintaxis.

Los métodos que la interfaz Visitor contiene suelen tener por signatura visitX(X ast, Object o), donde X es el tipo de árbol de sintaxis y el nombre de la clase que lo representa, por ejemplo: visitProgram(Program ast, Object o).



2. ¿Qué es IDE-Triangle?

IDE-Triangle es un pequeño entorno de desarrollo integrado (IDE) que permite al usuario escribir programas para Triangle, compilarlos, ejecutarlos y observar el código generado para TAM, los árboles de sintaxis abstracta (ASTs) y los detalles de la tabla de identificadores.

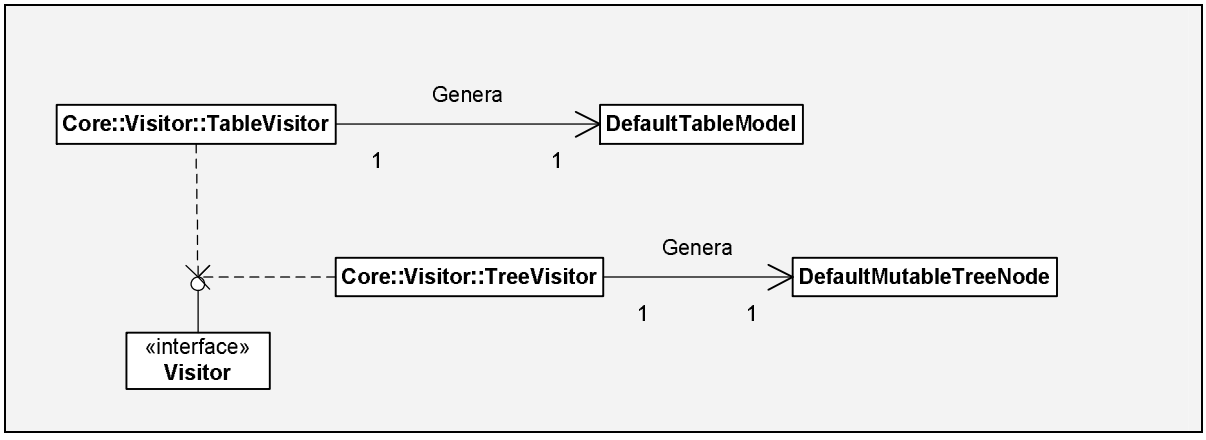
Al igual que las más recientes implementaciones del compilador de Triangle, IDE-Triangle fue desarrollado en Java con el propósito de conectar ambos componentes (compilador y entorno) sin mezclar sus funciones, permitiendo así que versiones modificadas del compilador puedan ser conectadas al IDE.

2.1. Cómo conectar el compilador a IDE-Triangle

Conectar una implementación de Triangle en Java a IDE-Triangle es un proceso fácil. Básicamente, lo único que se debe hacer es colocar el archivo .jar de Triangle, llamado Triangle.jar en el mismo directorio donde se encuentra el archivo .jar del IDE, llamado IDE-Triangle.jar.

Sin embargo, si se han hecho modificaciones en los árboles de sintaxis abstracta como adiciones de nuevas estructuras o cualquier tipo de cambio en la interfaz Visitor, debe hacerse también en algunas clases del IDE. Estas clases se encuentran en el paquete Core.Visitors del IDE y son implementaciones de la interfaz Visitor, utilizadas para recorrer los árboles de sintaxis.

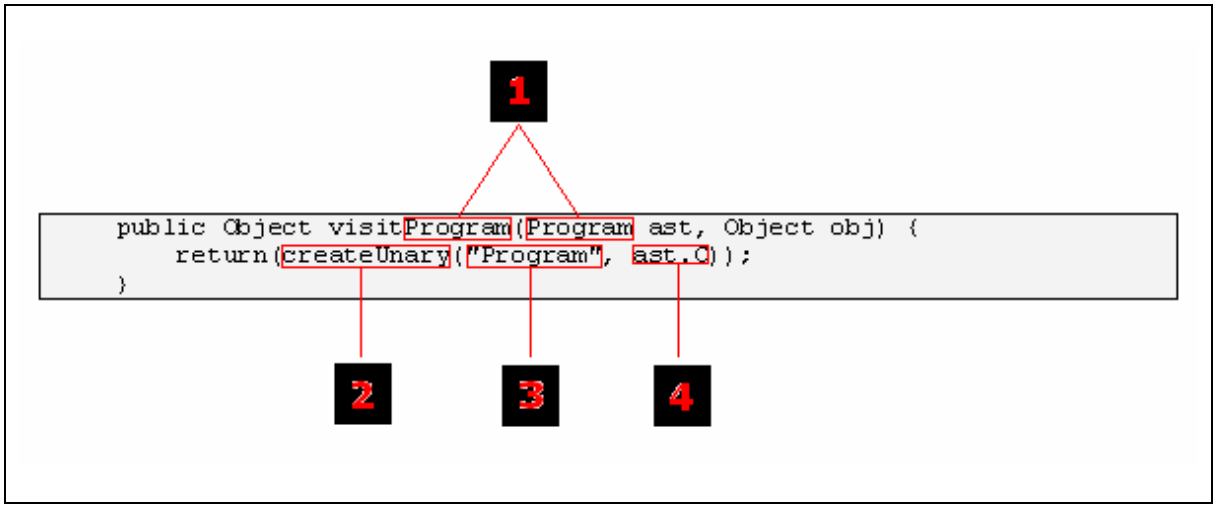
El siguiente diagrama de clases muestra las dos clases del IDE que implementan la interfaz Visitor.



2.1.1. La clase TreeVisitor

El propósito de la clase TreeVisitor es recorrer los árboles de sintaxis para generar un árbol de tipo DefaultMutableTreeNode y así generar un JTree para poder graficarlo en el IDE. Si se añadieron nuevos métodos a la interfaz Visitor del compilador, esos métodos se deben ver reflejados también en TreeVisitor.

Los métodos en la clase TreeVisitor tienen todos, por lo general, la siguiente signatura:



Donde:

1. El nombre de la clase que representa el árbol de sintaxis. Se suele seguir la misma convención con el nombre del método, colocándole la palabra “visit” adelante.

2. El tipo de árbol de sintaxis que representa esa clase, que puede ser Nullary, Unary, Binary, Ternary o Quaternary; dependiendo de las características de dicho árbol.

3. El *caption* o texto que aparecerá al momento de graficar el árbol.

4. Los hijos del árbol. La cantidad de hijos depende del tipo de arbol (si es Nullary es cero, si es Unary es uno…)

Por ejemplo, una estructura llamada Patito, la cual como *caption* mostraría “Un Patito” y es del tipo ternario con hijos llamados Ana, Pepe y Julio tendría un método de la siguiente forma:

```
public Object visitPatito(Patito ast, Object obj) {
    return(createTernary("Un Patito", ast.Ana, ast.Pepe, ast.Julio));
}
```

Hay algunas excepciones, como los identificadores u operadores, en los cuales se utiliza como *caption* una propiedad que traen los árboles llamada *spelling* que es un String con el nombre de la estructura. Por ejemplo:

```
public Object visitOperator(Operator ast, Object obj) {
    return(createNullary(ast.spelling));
}
```

2.1.2. La clase TableVisitor

El propósito de la clase TableVisitor es recorrer el árbol de sintaxis abstracta para generar la tabla de identificadores. Modificar esta clase es una tarea mucho más fácil ya que en algunos casos hay que escribir métodos que no hagan nada y retornen nulo, o que simplemente visiten a sus hijos.

Dependiendo del tipo de estructura que fue añadida al compilador, el método para el visitante será distinto. Por ejemplo, si es solamente un árbol binario llamado “Salto” con dos hijos X y Y que han de ser recorridos, lo único que iría en el método sería lo siguiente:

```
public Object visitSalto(Salto ast, Object o) {
    ast.X.visit(this, null);
    ast.Y.visit(this, null);

    return(null);
}
```

Mucho más fácil en el caso de los árboles Nullary, donde solo habría que escribir el siguiente método:

```
  public Object visitMyNullaryTree(MyNullaryTree ast, Object o) {
      return(null);
  }
```

Pueden darse casos en los que la nueva estructura tenga un nuevo identificador y este deba mostrarse en la tabla. Para ello, se debe llamar al método addIdentifier(), el cual es propio de esta clase e inserta en la tabla un nuevo identificador.

addIdentifier() recibe los siguientes parámetros:

- Name: Nombre del identificador, por lo general se usa la propiedad spelling.
- Type: Tipo del identificador.
- Size: tamaño que ocupa en memoria el identificador
- Level: nivel en el cual el identificador se encuentra
- Displacement: desplazamiento/*offset* del identificador en memoria.
- Value: valor por ***default*** (para constantes).

No todas las estructuras tienen nivel, desplazamiento o valor, por lo cual en esos casos se debe colocar un "-1" y no se mostrará nada en la tabla.

Supongamos que tenemos un árbol binario llamado "Foo", el cual contiene como hijos un identificador, llamado I, y otro árbol cualquiera llamado X. El tipo de identificador que se genera es una variable que el generador de código ve del tipo KnownAddress. Los identificadores del tipo KnownAddress tienen nivel y desplazamiento, más no un valor ***default*** pues no son constantes. En este caso, nuestro metodo sería así:

```
public Object visitFoo(Foo ast, Object o) {
      addIdentifier(ast.I.spelling, /* el nombre del identificador */
              "KnownAddress",  /* el tipo */
              ast.entity.size,  /* tamaño del identificador */
              ((KnownAddress)ast.entity).address.level, /* nivel */
              ((KnownAddress)ast.entity).address.displacement,  /*despl*/
              -1 /* valor, no se tiene */);

      ast.I.visit(this, null); // visitamos el primer hijo
      ast.X.visit(this, null); // visitamos el segundo hijo.
      return(null);
  }
```

El ejemplo anterior es una llamada típica a addIdentifier(). La información de un identificador la guarda la propiedad ***entity***. En algunos casos se recurre al polimorfismo

para poder acceder a propiedades y métodos propios de alguna estructura, como es el caso de KnownAddress.

Al modificar TreeVisitor y TableVisitor, se encuentra lista la conexión del IDE con el compilador. Sólo es necesario generar los archivos .jar correspondientes y colocarlos juntos en el mismo directorio para ejecutar el IDE.

2.1.3. Generación del Jarfile

Los archivos .jar son las librerías de Java. Internamente, son similares a un archivo .zip, aunque con algunas variaciones.

Es necesario generar un .jar del compilador de Triangle antes y después de hacer las modificaciones al IDE-Triangle. Antes para poder importar la librería utilizando cualquier IDE de preferencia como NetBeans o Eclipse y hacer las modificaciones al IDE-Triangle de modo que se pueda compilar satisfactoriamente y después para poder colocarlos en el mismo directorio.

Para generar el archivo Triangle.jar, se debe comprimir en un .zip lo siguiente:

- El directorio TAM con todos sus subdirectorios
- El directorio Triangle con todos sus subdirectorios
- Un directorio META-INF que contenga un archivo llamado MANIFEST.MF

El archivo MANIFEST.MF puede ser un simple archivo de texto con el siguiente contenido:

```
Manifest-Version: 1.0
Created-By: 0.92-gcc
```

Una vez generado el archivo .zip con el contenido mencionado anteriormente, se le debe cambiar la extensión de .zip a .jar, llamándose Triangle.jar.

Para hacer las modificaciones de TreeVisitor y TableVisitor, se debe importar un .jar actualizado como librería dentro de su editor de preferencia. Para ejecutar IDE-Triangle, se deben colocar los archivos IDE-Triangle.jar y Triangle.jar en el mismo directorio.

3. Información de contacto

Agradecería cualquier tipo de comentarios, sugerencias y reportes de errores sobre IDE-Triangle a la dirección de correo: **luiperpe** en el dominio **ns.isi.ulatina.ac.cr**

El sitio web oficial de IDE-Triangle es: http://ns.isi.ulatina.ac.cr/~luiperpe.