

Frank E. Pérez Rivero | Alex R. Coto Santiesteban | 2012-2013 | Marzo de 2013

TIGER

Informe del proyecto de compilación

# Introducción

El presente documento constituye el informe correspondiente al proyecto de Compilación de la asignatura Complementos de Compilación, impartida en la facultad de Matemática y Computación de la Universidad de la Habana. Dicho proyecto consiste en realizar una implementación funcional de un compilador para una versión modificada del lenguaje Tiger.

En el presente informe, pretendemos resumir de manera organizada y comprensible la arquitectura e implementación de nuestra implementación del compilador, así como algunos de los detalles que requirieron de especial atención por nuestra parte.

Comencemos mencionando algunos detalles generales acerca del proyecto.

# Generalidades

## Herramientas utilizadas

* **ANTLR/ ANTLRWorks** para la generación automática del analizador lexicográfico (en lo adelante lexer) y del analizador sintáctico (en lo adelante parser).
* **C#**/**Visual Studio**/**IL** como lenguaje de programación, entorno de desarrollo, y lenguaje de generación de código, respectivamente.
* **GIT**/**TortoiseGit**/**Beyond Compare** como herramientas de control de versiones.

## Metodología

Para realizar dicho proyecto, dividimos el mismo en las 3 etapas, que mencionamos a continuación:

1. Análisis lexicográfico y sintáctico
2. Análisis semántico
3. Generación de código

Estas etapas ocurren de forma ordenada, y solo se procede a la siguiente fase si la anterior concluyó satisfactoriamente. La información que resulta de cada etapa es aprovechada en la siguiente.

A partir de la gramática de TIGER descrita en la especificación del lenguaje facilitada por el colectivo de la asignatura, se generaron ambos lexer y parser. Fue necesario hacer algunas transformaciones a la gramática, con el objetivo de que ésta reflejara algunos elementos como la precedencia de operadores y otros, los cuales mencionaremos a su debido tiempo.

Posteriormente, a partir del lexer y el parser, obtenemos el árbol de sintaxis abstracta (**AST**) correspondiente al código que se pretende compilar. Dicho AST es utilizado en la fase 2 para tratar de encontrar errores semánticos y reunir información sobre los tipos de datos para la fase posterior de generación de código.

Finalmente, se genera el programa en código IL. El resultado del proceso de compilación consiste en un archivo binario ejecutable (**.exe**), cuyo contenido es ejecutado por el .NET Framework. No resulta objetivo del proyecto la implementación de una ulterior fase de optimización de código.

# Análisis lexicográfico y sintáctico

A pesar de que ambas etapas suelen verse por separado, hemos decidido mencionarlas como una única etapa en el compilador debido a que ANTLR permite generar a partir de un archivo “.g” ambos lexer y parser simultáneamente.

Se prescindió prácticamente por completo de embeber código en la gramática de ANTLR, con el objetivo de mantener durante todo el desarrollo del proyecto dicho archivo lo más legible posible, para facilitar (de ser necesarios) posteriores cambios al mismo.

La gramática del lenguaje facilitada por el colectivo fue prácticamente plasmada (con la debida sintaxis, por supuesto) en el archivo “.g” a partir del cual se generaron los mencionados componentes. Los cambios que fue necesario realizar en la misma, así como los detalles más interesantes los mencionamos brevemente a continuación:

## Precedencia de operadores

Para expresar la precedencia de los operadores en la gramática de ANTLR, la producción “expr” fue “dividida” (por decirlo de algún modo) en subsecuentes producciones que permitieran expresar este detalle de forma correcta. Fue comprobado exhaustivamente que los cambios realizados no provocaron un cambio en el lenguaje generado por dicha gramática, sólo en la estructura del AST generado. Los detalles se pueden observar fácilmente en el archivo “.g” adjunto a este documento.

## Bloques de declaraciones

Para facilitar la posterior fase de análisis semántico, específicamente en lo que respecta a las verificaciones correspondientes a los bloques de declaraciones (las cuales mencionaremos más adelante), añadimos producciones adicionales para que el parser agrupara declaraciones consecutivas de un mismo tipo en un nodo concreto del AST.

## LValue

Las producciones correspondientes al LValue fueron modificadas para que quedara reflejado el orden en que son aplicadas en el AST.

## Secuencias de escape en cadenas de texto

Las secuencias de escape fueron implementadas durante la fase de análisis lexicográfico utilizando la clase de .NET Regex. El procesamiento de dichas secuencias se realizó en el constructor de los nodos del AST para evitar la introducción de código fuente en la gramática de ANTLR. Dicho sea de paso, el constructor de los nodos del AST se ejecuta durante el chequeo lexicográfico.

## Comentarios

Para la implementación de los comentarios, fue necesario utilizar una opción de ANTLR: “greedy = false”. Esto garantiza…

## Ambigüedades

Si una cadena se puede producir mediante dos secuencias de producciones distintas, como es en ocasiones el caso de las operaciones aritméticas, el comportamiento de ANTLR por defecto consiste en utilizar la secuencia de producciones más corta e invalidar las restantes, comportamiento que, en nuestro caso, resultó suficiente para la implementación de lexer y parser.

En algunos casos, utilizamos predicados sintácticos para desambiguar algunas producciones, como la declaración y la instanciación de arreglos.

## Resultados del proceso

Como resultado del proceso de análisis sintáctico, se obtiene el ya mencionado árbol de sintaxis abstracta. Para especificar la forma del mismo, se utilizaron los operadores de reescritura de la gramática, que permiten especificar, a partir una producción, la forma que tendrá el sub-árbol resultante de dicha producción.

# Análisis semántico

Ya una vez construido el árbol de sintaxis abstracta, se procede a analizar la semántica del código fuente a compilar a partir del AST.

## Proceso general

Esta verificación se realiza de forma relativamente simple, gracias a las ventajas de la programación orientada a objetos aplicada a este problema. Los principios de abstracción, encapsulación, herencia y polimorfismo fueron aprovechados al máximo en la medida de lo posible.

El proceso para un nodo cualquiera, en general, consiste en analizar semánticamente primero a sus hijos, y posteriormente analizar la semántica del nodo en sí. En algunos casos, como en el caso de los bloques de declaraciones, es necesario realizar un procedimiento ligeramente distinto, que veremos un poco más adelante.

Durante esta etapa, se crean determinadas estructuras de datos con el propósito de utilizarlas, tanto para comprobar que el código fuente que se compila es semánticamente correcto, como para crear enlaces que luego se aprovecharán durante la fase final de generación de código.

## Ámbitos

Tiger es un lenguaje que utiliza “static scoping”. Algunas instrucciones, tales como los “let”, crean un ámbito interno (o hijo), desde el cual puede accederse a cualquiera de las variables, tipos y funciones declarados en un ámbito más externo (o padre). Un ámbito se crea siempre a partir de otro (excepto el primero, por supuesto), lo que garantiza que estos siempre estarán enlazados entre ellos de forma consistente.

Las tablas de símbolos de cada ámbito se implementaron utilizando tablas hash. Decimos “las tablas”, y no “la tabla”, porque aunque las variables y las funciones comparten el mismo espacio de nombres, los tipos tienen uno distinto que les resulta propio.

Para resolver un nombre, un ámbito determinado busca primero en su tabla de símbolos, y luego recursivamente en las tablas de sus ámbitos padres en caso de no contener dicho símbolo. Este mecanismo garantiza una implementación simple del mecanismo de ocultación de nombres, ya que simplemente se devuelve la primera estructura cuyo símbolo en la tabla coincida con el símbolo que se busca. Al no ser posible declarar dos símbolos dentro del mismo ámbito con el mismo nombre, se garantiza que el mecanismo de ocultación funciona siempre en el orden correcto. Otro detalle que es importante destacar es que hay algunos símbolos que no es posible redefinir (una suerte de “sólo lectura”), como las funciones predefinidas del lenguaje.

Durante esta etapa, cada identificador se enlaza con lo que representa en el ámbito actual. Inicialmente sólo enlazábamos los nodos con sus correspondientes ámbitos, y resolvíamos los nombres utilizando el ámbito asociado a este, pero esto resultaba un problema cuando, por ejemplo, declarábamos una variable con un nombre de una ya existente (la nueva ocultaría a la anterior), pero en la inicialización de esta, utilizábamos a la variable con el mismo nombre que existía en algún ámbito padre. Esto representaba un problema en la fase de generación de código, ya que la nueva variable, al haber sido agregada al ámbito, se trataba de inicializar utilizándose a ella misma.

El ámbito global se inicializa también durante esta fase. Se agregan todas las funciones predefinidas del lenguaje como símbolos de sólo lectura para evitar la pérdida de funcionalidades por ocultación, así como los tipos de datos primitivos: números enteros y cadenas de texto.

## Tipos de datos

En el análisis semántico, también realizan verificaciones respecto a los tipos de datos utilizados.

### Definiciones de tipos

En primer lugar, se pueden definir nuevos tipos: records y arrays, así como alias para estos tipos. Para verificar que no haya circularidad en estas declaraciones de tipos, al declararse los tipos en un bloque de declaraciones, se agregan simplemente al ámbito en el cual están siendo declarados, con los nombres de los tipos a los que referencian almacenados como simples cadenas de texto. Luego de que todos han sido agregados, se procede a “resolver” cada una de estas referencias. Se hace entonces una especie de búsqueda en profundidad, donde se marca, cada vez que se intenta resolver un tipo, los nodos por los que ya se ha transitado durante la resolución del tipo actual. Si en algún momento, nos encontramos con un tipo por el cual estamos transitando, es que existe un ciclo en las declaraciones. Los tipos que ya se han resuelto (o que se encuentran dentro de un ciclo) se marcan de una forma distinta para indicar que no es necesario transitarlos de nuevo. Cabe agregar que dichas verificaciones de “ciclismo” sólo se realizan en el caso de los arrays y los alias: no representa un problema el caso de los records.

En segundo lugar, es necesario realizar verificaciones respecto a los tipos de las expresiones. Una expresión dada puede tener, como tipo de retorno, cualquiera de los tipos primitivos del lenguaje, uno de los tipos que hayan sido definidos en el programa, o los tipos especiales Void y Unknown.

### Tipos especiales

Void se utiliza cuando dicha expresión no deber devolver valor alguno. Se realizan las verificaciones pertinentes para que una expresión de este tipo no aparezca en donde se espera una expresión con valor y viceversa.

Unknown se utiliza cuando se desconoce (temporalmente) el tipo de la expresión que se está analizando. Este tipo surge cuando nos encontramos el valor nil, el cual puede ser asignado a varios tipos de datos a su vez (records, arrays y cadenas de texto), por lo que claramente, el tipo que representa no puede ser determinado inmediatamente. No representa un tipo en sí, es meramente un tipo auxiliar durante el chequeo semántico.

### Inferencia de tipos

Por último, es posible realizar inferencia de tipos en las declaraciones de variables, siempre y cuando la expresión de inicialización tenga un tipo bien determinado (i.e. cualquiera menos Unknown), en cuyo caso, el tipo debe ser especificado explícitamente, además de aceptar el valor nil.

# Generación de código