

**Procesadores de Lenguajes**

**Generación de Código**

**Docente**: Cristian Prats

**Integrantes**:

Javier Villagra

Santiago Russo

Fecha: 16/12/2013

Versión: 1.0

Tabla de Revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Descripción | Autor |
| 16/12/2013 | 1.0 | Creación del Documento | Javier Villagra, Santiago Russo |

Contents

[I: Introducción 7](#_Toc375084677)

[II: Patrones de Traducción 7](#_Toc375084678)

[2.1 Generación de Código y Estructura del Árbol 7](#_Toc375084679)

[2.2 Variables y Temporales 8](#_Toc375084680)

[2.3 Manejo de Variables en el Llamado a Funciones 8](#_Toc375084681)

[III: Optimización de Código 8](#_Toc375084682)

[V: Pruebas 9](#_Toc375084683)

[VI: Consideraciones Generales 9](#_Toc375084684)

[VI: Conclusiones 9](#_Toc375084685)

[VII: Conclusiones Generales 10](#_Toc375084686)

# I: Introducción

El siguiente documento explicara a grandes rasgos el proceso de generación de código fuente a partir del árbol obtenido como salida del analizador semántico. Adjunto se encontrara un CD con el código fuente, ejecutable y pruebas.

# 

# II: Patrones de Traducción

Como se ha mencionado en la sección de Introducción, para la generación de código de ensamblado, se tomó como punto de partida el árbol generado en la etapa de análisis semántico.

Recordamos que se había diseñado una estructura conformada por nodos y hojas. En la parte de implementación se decidió generar una interfaz común a todos los nodos, para alinear a todas las clases involucradas bajo un mismo comportamiento. Estamos hablando de la interfaz “INodo”.

En esta etapa, se agregó a la citada interfaz el método “generateCode()”. La idea es que este método sea redefinido en cada tipo de nodo y genere, mediante la ayuda del “ExecutionTree”, el código assembler necesario, valiéndose de los propios atributos del nodo y de la tabla de símbolos.

## 2.1 Generación de Código y Estructura del Árbol

Para la generación de código maquina no ha sido necesaria una modificación en la estructura del árbol, tanto en sus clases Nodo como en sus Hojas. Simplemente se implementó en cada una de estas el método “generateCode()”.

A partir del nodo “S”, el cual representa la raíz del árbol, se comienzan a invocar en forma encadenada todos los “generateCode()” a medida que se va recorriendo el árbol en “Pos Orden” (Hijo izquierdo, hijo derecho y por último el nodo). Desde el “generateCode()” de cada Nodo, se invoca al “ExecutionTree” el cual es el encargado de ir copiando el código maquina generado dentro de un archivo de extensión ASM. También el “generateCode()” se vale de la información de la tabla de símbolos para, por ejemplo, recuperar el contexto de alguna variable, o las declaraciones que se realizaron en algún determinado contexto, y saber así de esta manera si el parámetro fue pasado por valor o por referencia.

El “ExecutionTree” se encarga de preparar el archivo ASM para ir recibiendo las instrucciones desde cada Nodo que lo invoque. En esta preparación, no solo genera un archivo ASM sino que también incluye instrucciones de cabecera para el correcto funcionamiento del archivo final dentro del “emu8086”. Dentro de este “header” se ha incluido la librería emu8086.inc para realizar las lecturas de teclado y los output por pantalla.

La salida del proceso de generación de código se puede encontrar en la carpeta “/output” con la extensión .asm y con el mismo nombre del archivo LM2 de código fuente HLC.

Cabe destacar que a la hora de generar tanto las variables enteras como las booleanas, se definieron ambas con DW, si bien entendemos que esta decisión no es de lo más performante, la cantidad de líneas de código generado baja drásticamente. Por otro lado todos los “PROC” fueron declarados como NEAR.

## 2.2 Variables y Temporales

Para evitar la generación de código erroneo (cuestión que nos dimos cuenta luego de varias horas de prueba), se ha decidido recorrer primeramente los NodosProc y NodoFunc correspondientes y generar al principio de toda ejecución la declaración de Variables tanto globales como de cada función en particular.

Para el manejo de temporales se utilizó la pila. Los mismos son almacenados en la pila para luego ser accedidos.

## 2.3 Manejo de Variables en el Llamado a Funciones

En el llamado a funciones se ha decidido guardar en la pila los valores de las funciones y temporales correspondientes anteriores a un llamado. Esto se realizó para evitar la modificación de sus valores durante los llamados recursivos.

Luego del llamado a una función, el valor de retorno queda almacenado en AX, por lo tanto se almacena dicho valor y se comienza a obtener nuevamente los valores que tenían antes de llamarse a esa función las variables y temporales correspondientes (como los temporales están en la pila, lo único que se hace es correr el stack pointer). Luego así, se sigue operando normalmente.

Los parámetros que son pasados por referencia a dicha función no son almacenados en la pila ya que se necesita que su valor sea modificado.

# III: Optimización de Código

El uso indiscriminado de variables temporales dentro del código generado en assembler puede ocasionar problemas de performance debido al uso indiscriminado de recursos.

Es por este motivo que cualquier nodo que necesite emplear una variable temporal, debe pedírselo a la clase “TemporalManager”, el cual no solamente otorga la posición dentro de la pila a utilizar para un determinado temporal. De esta forma no se asigna memoria de manera innecesaria y la posición de la pila es recuperada en cuanto el temporal deja de necesitarse.

También se tuvieron en cuenta los siguientes casos a la hora de otorgar una variable temporal. En una expresión, el nodo que está a la izquierda no pide un temporal al menos que estrictamente lo necesite ya que el nodo que está en la derecha va a emplear directamente el registro AX. Y de forma análoga el nodo que está a la derecha tampoco requerirá de una variable temporal si puede realizar la operación directamente en AX, este es el caso en que sea una hoja.

Otro tema fue la declaración de constantes. Para evitar realizar dos pasadas por el código generado para reemplazar las invocaciones a constantes por su valor estático, se optó por directamente insertar en valor de la misma cuando el “generateCode()” se topa con alguna, tomando el valor de la tabla de símbolos.

# V: Pruebas

Las pruebas realizadas se encuentras anexas en la carpeta Test. Se ha demostrado que el Código ejecuta Pruebas de Hanoi hasta 15, ya que luego el valor de cálculo supera los 2 bytes y ocurre overflow. Además, se han probado pruebas de generación de la serie de Fibonacci de forma recursiva hasta el número 15.

# VI: Consideraciones Generales

Se ha tenido en cuenta que a la hora de acceder al valor de un Array mediante un índice de tipo entero, el valor de este se encuentre entre 0 y el tamaño máximo del Array. En caso contrario, el programa devuelve la leyenda “Index out of bounds” o “Index non Positive” y termina con la ejecución.

# VI: Conclusiones

La generación de código fue mucho más compleja de lo esperado. Pensábamos que solo iba a ser transcribir código assembler según sea necesario dependiendo del nodo, pero no se tuvieron en cuenta muchas otras cuestiones. También complico un poco las cosas no saber codificar en assembler, lo cual hacia muy duras las pruebas cuando se encontraban errores y no se terminaba de dimensionar bien el problema. Por otro lado, el emulador a veces no se comportaba de una manera amigable, aún nos queda la incertidumbre si esto se debió a nuestra limitada pericia en el tema o a que la calidad del mismo no es de la mejor.

A pesar de estos inconvenientes, encontramos interesante el tema de acercarnos un poco más al bajo nivel de máquina y ver como suceden las cosas desde un punto de vista más ligado al hardware.

# VII: Conclusiones Generales

Encontramos la temática de la materia por demás interesante. Consideramos que aunque el resultado del trabajo que realizamos no sea de lo más útil (ninguno de nosotros se va a poner a programar empleando este lenguaje tan limitado) nos ayudó a dimensionar y terminar de entender que es lo que sucede cuando se emplea un lenguaje de alto nivel a la hora de bajarlo a la realidad de la máquina. Veníamos con un conocimiento teórico de Lenguajes Formales, pero esta materia ayuda a bajar esas ideas desconexas, a veces, a la realidad.