# Diseño de Compiladores

## Trabajos prácticos 3 y 4

Grupo 2

Mastieri, Julian: julimastieri@gmail.com

Goicoechea, Maria Pia: maripigoico@gmail.com

# Contenidos

**GENERACIÓN DE CÓDIGO INTERMEDIO** ..... **3**

USO DE LA NOTACIÓN POSICIONAL DE YACC ..... 4

**GENERACIÓN DE CÓDIGO ASSEMBLER** ..... **4**

INDICAR ..... **¡ERROR! MARCADOR NO DEFINIDO.**

**MODIFICACIONES DE ETAPAS ANTERIORES** ..... ¡ERROR! MARCADOR NO DEFINIDO.

**DESARROLLO DE COLECCIONES CON INFERENCIA** ..... **5**

**CONCLUSIONES** ..... **5**

# Temas particulares

Los temas particulares asignados al grupo son los siguientes:

Análisis léxico:

- Enteros largos sin signo: Constantes enteras con valores entre 0 y $2^{32} - 1$.
- Palabras reservadas foreach e in.
- Incorporación a la lista de tokens los literales "[" y "]"
- Comentarios de 1 línea: Comentarios que comienzan con # y terminan con el fin de línea.
- Cadenas multilinea: Cadenas de caracteres que comienzan con "{" y terminan con "}".

Análisis sintáctico:

- Iterador para colecciones (foreach).
- Colecciones con inferencia.
- Conversiones Explicitas: to_ulong.

Análisis semántico ( generación de código intermedio) :

- Representación mediante Árbol Sintáctico.
- Chequeo de compatibilidad de tipos: solamente se permiten operaciones con operandos de distinto tipo si se efectúan las conversiones explicitas correspondientes.

Generación de código Assembler:

- Seguimiento de registros.

# Introducción

En esta segunda entrega se realizó el chequeo semántico, creando código intermedio con la representación de árbol de parsing. Además, se generó código Assembler para Pentium de 32 bits a partir de la representación código intermedio. La representación asignada para dicho código fue la de árbol sintáctico.

Para la producción del código Assembler se utilizó el mecanismo de Seguimiento de registros mediante una tabla de ocupación de registros. Este mecanismo permite generar un código mas corto y rápido que el que se genera con el mecanismo de Variables Auxiliares.

El compilador devuelve cuatro archivos, los cuales corresponden a los errores detectados, el árbol sintáctico, la Tabla de Símbolos y el código Assembler correspondiente.

# Generación de código intermedio

La estructura dinámica utilizada para el almacenamiento del código intermedio fue el Árbol sintáctico.

Se genera código intermedio para todas las sentencias ejecutables, incluyendo: Asignaciones, Selecciones, Sentencias de control, Foreach, y Sentencias print.

Se debió incorporar a la tabla de símbolos atributos como el tipo de dato, su uso el cual puede ser contante, variable o nombre de colección entre otros; así como también los valores iniciales en el caso de declarar una colección a la que se le especifiquen los mismos. El tamaño de cada variable también se especificó, por la necesidad de conocer el tamaño de una colección.

El tipo y uso de las variables se registra a partir de las sentencias declarativas, mientras que en el caso de ser contantes el tipo y uso se definen en el análisis léxico. En el caso de las constantes negativas, se registrará su uso y tipo durante el análisis sintáctico.

El tamaño de una colección se completa con valor indicado explícitamente o, en el caso de una lista de valores iniciales, se infiere a partir de la cantidad de elementos de dicha lista. Cuando se incluyan valores de inicialización en la declaración de una colección, los mismos quedan registrados como valores iniciales de dicha colección.

En este punto, se adicionó un nuevo error el cual consiste en que no se permita valores iniciales incompatibles con el tipo de la variable declarada. Por ejemplo, si la colección es de tipo ulong, sus valores iniciales deben ser de este tipo y no enteros; de lo contrario sino se permitiría que este tipo de variable tenga valores negativos impropios a su tipo.

En cuanto a los **chequeos semánticos** que se detectan e informan como error se encuentran:

- Variables y colecciones **no declaradas**, es decir, que si no se le ha asignado un tipo a esa variable significa que no ha aparecido entre las sentencias de declaración que es donde se le asigna el tipo.
- Variables y colecciones **redeclaradas**, siguiendo la misma lógica, si ya se le ha asignado un tipo, significa que ya se declaró anteriormente. No se permite el uso de un mismo nombre para una colección y una variable, cada nombre de variable es único independientemente de su uso o tipo.
- Chequeos de **compatibilidad de tipos**, en este caso al existir la posibilidad de realizar conversiones explícitas de “int” a “ulong “, solo se permiten operaciones entre tipos distintos si se realiza la conversión correspondiente. Dentro de las operaciones se incluyen: asignación, comparación y operaciones aritméticas. Vale aclarar que, en el caso de las constantes que pertenecen al rango de los enteros, se le asigna dicho tipo, por lo que si se le quisiese asignar a una variable ulong se deberá realizar la conversión correspondiente (to_ulong(2)).
- No se permite el nombre de una variable usada colección, ni el nombre de una colección usado como variable.
- Se corrobora que el subíndice de un elemento de colección sólo podrá ser una variable o constante de tipo int.
- En cuanto al iterador para colecciones el bloque se ejecutará tantas veces como lo indique el tamaño de la colección. Se controla que la variable utilizada para iterar tenga dicho uso y lo

mismo para la colección, además de que se chequea que ambos sean del mismo tipo sino habrá un error.

### Uso de la notación posicional de Yacc

En los párrafos subsiguientes se detallará de qué forma se utilizó a notación posicional de Yacc. En primer lugar, se utilizó para construir el árbol sintáctico. Cada vez que genera un nodo del árbol, el mismo se asigna a la variable $$ y de esta forma se logra construir el árbol desde las hojas hasta la raíz. Si fuese necesario al nodo, antes de asignarlo, se le configuran diversas variables del mismo, que luego esta información es utilizada en la generación del código assembler. Estas variables pueden ser el tipo y nombre, como también el hijo izquierdo y el derecho que realizan la construcción del árbol.

Como segunda utilidad de "$$", fue para construir una lista con los valores iniciales de una colección y así poderlos almacenar en el token; de la misma manera se usa en las declaraciones de variables en donde se devuelve una lista de variables declaradas para posteriormente asignarles el tipo.

Con respecto a la notación "$n" fue usada para referenciar a nodos hijos construidos en sentencias previas. También se utilizó como lexema para modificar los tokens de la tabla de símbolos y modificar atributos como el tipo, uso, valores iniciales y demás. Para los chequeos semánticos mencionados anteriormente también se utiliza esta notación.

Para generar las bifurcaciones de las sentencias de control se introduce, durante la generación del árbol, un nodo con el nombre de "CONDICION" o "CONDICION_FOREACH. En cada ocurrencia de estos nodos se debe generar las sentencias de control en assembler.

## Generación de código Assembler

Para la generación de código assembler se implemento un buffer donde se van concatenando las instrucciones indicadas para cada sentencia para luego generar como salida un archivo asm que contiene el programa completo traducido a código assembler.

El mecanismo utilizado fue la generación mediante seguimiento de registros. Para esto, se tiene un arreglo de cuatro casilleros, los cuales representan cada uno de los registros, para llevar un control de los libres y ocupados en cada momento.

Para el caso de las operaciones aritméticas, se distinguieron cuatro casos o situaciones generales. Estos diferencian entre las operaciones entre:

- Dos variables y/o constantes y/o elementos de colecciones. En este caso se genera código sobre un nuevo registro a ocupar.
- Un registro y una variable o constante o elemento de colección. Simplemente se genera código sobre el registro existente, sin ocupar nuevos registros.
- Dos registros, generando código sobre el primer registro mientras que se libera el segundo.
- Una variable, constante o elemento de colección y un registro. Para esta situación se debió tener en cuenta si la operación que se va a traducir es conmutativa (como suma y multiplicación) o no es conmutativa (como resta y división).
  En el caso de ser conmutativa, se genera código sobre el registro (operando derecho), sin ocupar nuevos registros.

Mientras que si no es conmutativa, se ocupa un nuevo registro para almacenar la constante, variable o elemento de colección y se genera código sobre este. A su vez, se libera el primer registro.

En las operaciones con operandos de tipo ulong, se utilizaron los registros EAX, EBX, ECX y EDX. Mientras que en los casos que para las que poseen operandos de tipo entero se utilizaron AX, BX, CX y DX.

Además, en el caso de la multiplicación y división se utilizaron las instrucciones IMUL / IDIV para el caso de los operandos enteros y las instrucciones MUL / DIV para el caso de operandos ulong. Esto es ya que IMUL e IDIV trabajan teniendo en cuenta el signo de los operandos y del resultado. Por otra parte, MUL y DIV realizan las operaciones sin tener en cuenta el signo, resultando útiles para operandos ulong que no pueden ser números negativos.

Durante esta etapa también se tuvieron en cuenta **chequeos** que deben ser evaluados en **tiempo de ejecución**.
Por una parte, al realizar las divisiones se verifica que no se realicen **divisiones por cero**. Esto es, en caso de que el divisor sea cero, el programa emitirá un mensaje de error y finalizará su ejecución.

Por otra parte, también se verifica el caso que se produzca perdida de información al realizar conversiones implícitas. Esto se da únicamente cuando se realiza una conversión *to_ulong* sobre un entero negativo. Al igual que en el caso anterior, se emite un mensaje de error y se finaliza la ejecución del programa haciendo un salto a la etiqueta LabelError.

Además, se controla el valor de los subíndices de las colecciones, verificando que únicamente se utilicen variables o constantes de tipo entero.

## Desarrollo de colecciones con inferencia

Para poder inferir el tamaño de las colecciones, se realizó una acción dentro de la regla de la gramática donde se declara la colección junto con sus valores iniciales.

Dentro de esta acción, se utiliza un método que utiliza el nombre o id de la colección y los valores iniciales en forma de lista de Strings. Así, se agrega en la tabla de símbolos información sobre la colección especificada, como el tamaño (size de la lista) y los valores iniciales (se asignan los valores de la lista recibida).

## Conclusiones

En conclusión, se puede decir que el compilador completo funciona correctamente, es decir, las cuatro etapas involucradas (análisis léxico, sintáctico, semántico y traducción) cumplen sus respectivas funciones.

En el caso de que se produzca algún error durante alguna de las primeras tres etapas, este será informado y no se generará el código assembler correspondiente hasta que no se solucione el/los error/es en cuestión.

Esto se puede evidenciar al ejecutar los distintos ejemplos provistos que abarcan todas los tipos de sentencias aceptadas por el compilador, desde asignaciones, operaciones aritméticas hasta sentencias de control y foreach.