# PRÁCTICA #7: Generación de Código Intermedio (Tipos Estructurados[1]).

## Objetivo.

El objetivo de esta práctica es ampliar el lenguaje *sampleC*[2] de manera que admita la definición y manipulación de arrays multidimensionales generando una representación intermedia.

## Preparación del Laboratorio.

Leer en el capítulo dedicado a la generación de código intermedio de Aho el párrafo dónde se explica la traducción de estructuras de tipo array (páginas 495–502). Esta práctica se ha de implementar a partir del laboratorio en el que se genera código intermedio para las *expresiones*.

## Descripción.

Las reglas de producción que definen la declaración de objetos de datos en la gramática de *sample C* se modifican de la siguiente manera:

program ::= definition {definition}

definition ::= dataDefinition | functionDefinition

dataDefinition ::= type declarator { "**,**" declarator } "**;**"

declarator ::= "**ID**" | "**ID**" "**[**" "**CONST**" "**]**" { "**[**" "**CONST**" "**]**" }

type ::= "**INT**" | "**LONG**"

Se propone la siguiente definición dirigida por la sintaxis para manipular las declaraciones de arrays[3]:

| Producción | Reglas Semánticas |
|---|---|
| $D \rightarrow T\ L$ | L.inherit := T.type |
| $T \rightarrow$ **INT** | T.type := INT.type |
| $T \rightarrow$ **LONG** | T.type := LONG.type |
| $L \rightarrow L_1$ **, I** | $L_1$.inherit := L.inherit<br>I.inherit := L.inherit |
| $L \rightarrow I$ | I.inherit := L.inherit |
| $I \rightarrow I_1$ **[ CONST ]** | $I_1$.inherit := array( CONST.value, I.inherit) |
| $I \rightarrow$ ID | addType(ID.entry, I.inherit) |

[1] Fisher,Charles N.; LeBlanc, Richard J., "Crafting a Compiler with C", Benjamin/Cummings, 1991.

[2] Schreiner, Axel T.; Friedman, H. George; "Introduction to Compiler Construction with UNIX"; Prentice–Hall, Inc., 1985

[3] Aho, A.; Sethi, R.; Ullman, J., "COMPILERS. Principles, Techniques, and Tools", Adisson–Wesley Publishing company, 1986.

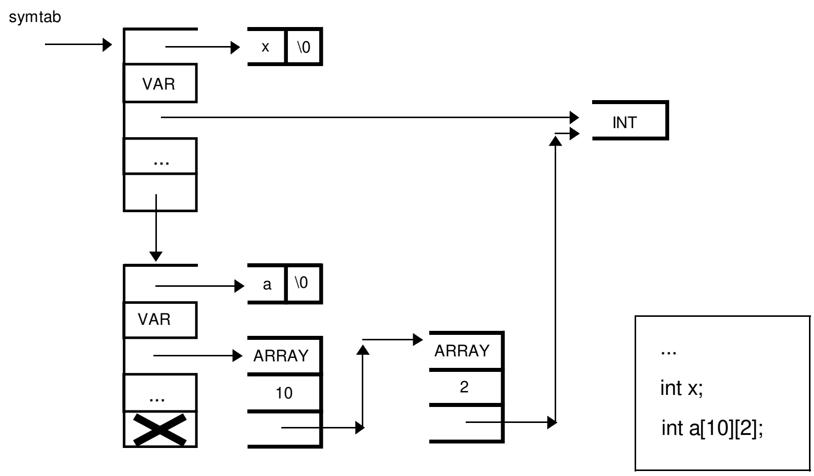
En esta definición dirigida por la sintaxis se asocia a la variable sintáctica *T* el atributo sintetizado *type* que almacena el tipo de la declaración. La variable sintáctica *L* tiene asociado un atributo heredado *inherit* mediante el cual se trasmite a los identificadores el tipo del cual han sido declarados.

Se trata de reconocer arrays multidimensionales al estilo del lenguaje C, por lo tanto sólo es necesario especificar el límite superior de cada dimensión puesto que por defecto el límite inferior en cualquier caso es cero. La variable sintáctica *I* se encarga de ir evaluando las cotas y con esa información crea el tipo estructurado de datos que se está definiendo. El atributo *inherit* asociado a *I* trasmite al identificador que se está declarando el tipo de datos que ha reconocido.

Nótese que la definición dirigida por la sintaxis anterior no tiene atributos por la izquierda pues para evaluar el atributo heredado *inherit* asociado a *I* es necesario conocer el valor de la constante (CONST) que aparece a su derecha en la regla de producción.

Puesto que con esta modificación del lenguaje se admiten tipos estructurados de datos, es necesario crear un Sistema de Tipos. El sistema de tipos que nos ocupa contará con expresiones de tipo para los tipos simples "integer" y "long" y para un único tipo estructurado "array". Se representará las expresiones de tipo mediante estructuras dinámicas de datos. Por lo tanto los atributos *type* e *inherit* se definen como punteros a la expresión de tipo que representan.

La función *array(CONST.value, I.inherit)* devuelve un puntero a una representación mediante una estructura de datos de una expresión de tipo. En dicha expresión se almacena el límite superior del array y el tipo del que han sido declarados sus elementos.




**Ejercicio 1.**

Ampliar el analizador sintáctico de *sample C* de manera que admita la definición de arrays multidimensionales.

Las reglas de producción que definen el acceso a objetos de datos en la gramática de *sampleC* se modifican de la siguiente manera:

```
binary  ::= variable "="  binary
          | variable "+=" binary
          | variable "−=" binary
          | variable "*=" binary
          | variable "/=" binary
          | variable "%=" binary
          | binary  "+"  binary
          | ...
          | unary
unary   ::= "++" variable
          | "−−" variable
          | primary
primary ::= variable
          | "CONST"
          | "(" expression ")"
          | "ID" "(" [argumentList] ")"
variable ::= "ID"
          | "ID" "[" binary "]" { "[" binary "]" }
```

Antes de generar código intermedio asociado a las expresiones y a la sentencia de asignación, se tiene que realizar la comprobación de tipos; en este sentido, se propone la siguiente definición dirigida por la sintaxis [3]:

| Producción | Reglas Semánticas |
|---|---|
| $E \rightarrow$ **ID** | E.type := ID.type |
| $E \rightarrow$ **CONST** | E.type := integer |
| $E \rightarrow E_1$ **+** $E_2$ | E.type := if $E_1$.type = integer and<br>$E_2$.type = integer then integer<br>else typeError |
| $E \rightarrow E_1$ * $E_2$ | E.type := if $E_1$.type = integer and<br>$E_2$.type = integer then integer<br>else typeError |
| $E \rightarrow E_1$ **[** $E_2$ **]** | E.type := if E2.type = integer and<br>$E_1$.type = array(upper,type) then type<br>else typeError |

El atributo sintetizado *type* para *E* da la expresión de tipo asignada por el sistema de tipos a la expresión generada por *E*. *ID.type* se utiliza para traer el tipo guardado en la tabla de símbolos asociado a *ID*, y se le asigna al atributo *type* asociado a *E*. La expresión de tipo que se crea al aplicar cualquier operador binario a dos subexpresiones de tipo entero tiene tipo entero, de lo contrario su tipo es un tipo error. En una referencia a un array la expresión $E_2$ deber tener tipo entero, en cuyo caso el resultado es el tipo del elemento (*type*) obtenido del tipo *array(upper, type)* de $E_1$.

Considere expresiones como x+i donde x es de tipo long e i de tipo entero. Como la representación de enteros y enteros largos es distinta dentro de una máquina, y se utilizan instrucciones distintas para las operaciones sobre enteros y enteros largos, puede que el compilador tenga que convertir primero uno de los operandos del + para garantizar que ambos operandos sean del mismo lipo cuando tenga lugar la suma.

La definición del lenguaje especifica las conversiones necesarias. Cuando un entero se asigna a un entero largo o viceversa, la conversión es al tipo del lado izquierdo de la asignación. En expresiones, la transformación más común es la de convertir el entero en un entero largo y después realizar una operación larga con el par de operandos long obtenidos. Se puede usar el chequeo de tipos en un compilador para insertar estas operaciones de conversión en la representación intermedia del programa fuente. Por ejemplo, la notación postfija para x + i puede ser:

**x i INT2LONG LONG+**

El operador **INT2REAL** transforma i de entero a long y después **LONG+** realiza la suma entero largo con sus operandos.

La siguiente definición dirigida por la sintaxis muestra las reglas de coerción de entero a tipo long.

| Producción | Reglas Semánticas |
|---|---|
| E → **CONST** | E.type := long |
| E → **ID** | E.type := ID.type |
| E → $E_1$ **Op** $E_2$ | E.type := if $E_1$.type = integer and $E_2$.type = integer then integer<br>else if $E_1$.type = integer and $E_2$.type = long then long<br>else if $E_1$.type = long and $E_2$.type = integer then long<br>else if $E_1$.type = long and $E_2$.type = long then long<br>else typeError |

## Ejercicio 2.

Ampliar el analizador sintáctico de *sample C* para traducir a código intermedio arrays multidimensionales. Añada el conjunto de instrucciones que considere necesario.