Enunciado General de la Práctica y Análisis Sintáctico

Curso 2020-2021 Versión 4

Jorge Bernal & F. Javier Sánchez

24/02/2021

Enunciado de la práctica de Compiladores para las entregas de sintáctico, semántico y generación de código.

1. Tabla de contenido

2. Introducción

Este documento sirve de guía para enfocar la primera sesión práctica de la asignatura Compiladors: Análisis Sintáctico, semántico y generación de código. Los objetivos de este documento son:

- Introducción del entorno en el cual se realizarán las prácticas.
- Primeros pasos y comandos útiles en CoSeL.
- Presentación de la gramática del lenguaje orientado a objetos simples (LOOS).
- Planteamiento de las ampliaciones de LOOS que habrán de ser implementados en CoSeL para la primera sesión de prácticas.
- Entregas.
- Evaluación.

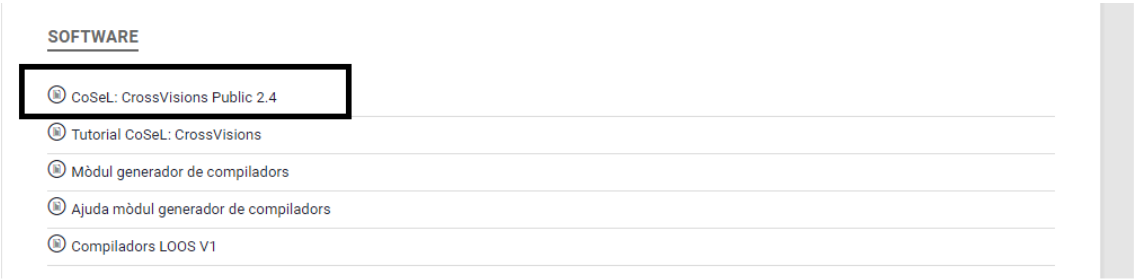
Este documento puede sufrir actualizaciones durante el curso. Estas se colgarán en el aula Moodle de Compiladors.

3. Primeros pasos con CoSel

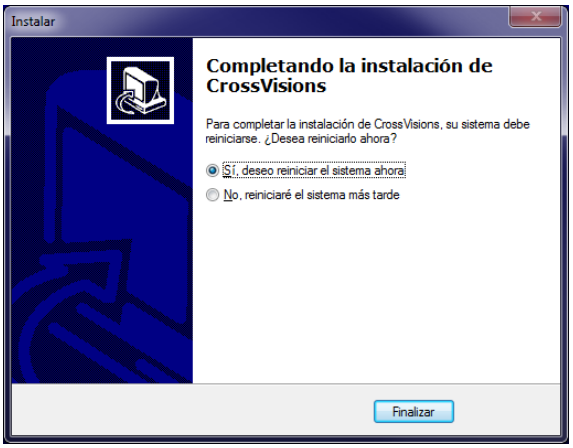
Aunque en este apartado se hace un pequeño resumen sobre CoSeL, es muy importante leer el tutorial de CoSeL que encontrareis en el fichero CrossVisions.csm. Además, en este fichero encorareis muchas de las funciones de CoSeL con su correspondiente explicación.

3.1. Instalación de CrossVisions

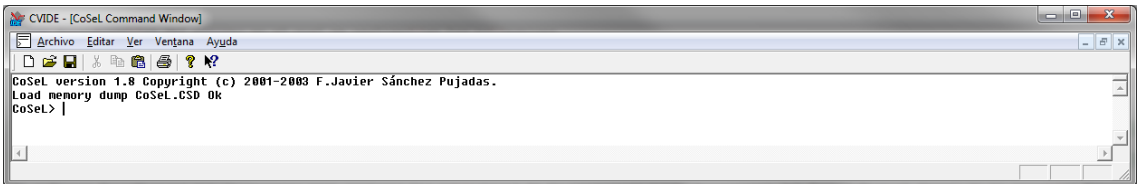
El software CrossVisions que se necesita para realizar las prácticas de la asignatura puede descargarse del Aula Moodle de la asignatura, dentro del apartado Software, tal y como se puede ver en la Figura:



El programa CrossVisions es compatible con cualquier versión de Windows desde Windows 7. Una vez descargado el ejecutable bastará con una simple instalación haciendo doble click en el fichero 'CrossVisions Public 2.4 - Setup.exe'. El proceso de instalación no presenta ninguna complicación y simplemente se ha de indicar al instalador que continúe cuando nos pregunte. Si la instalación ha finalizado correctamente se pedirá al usuario que reinicie el equipo, como se ve en la Figura:

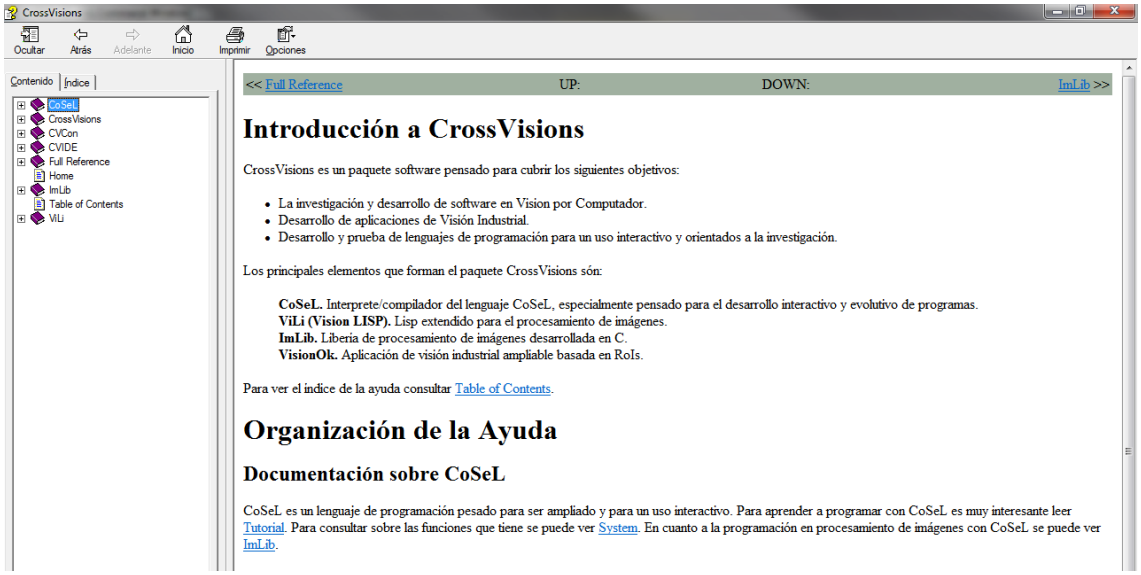


Para iniciar CoSeL simplemente basta con teclear CoSeL en el menú de Inicio de Windows. La pantalla inicial que veremos es la siguiente figura:

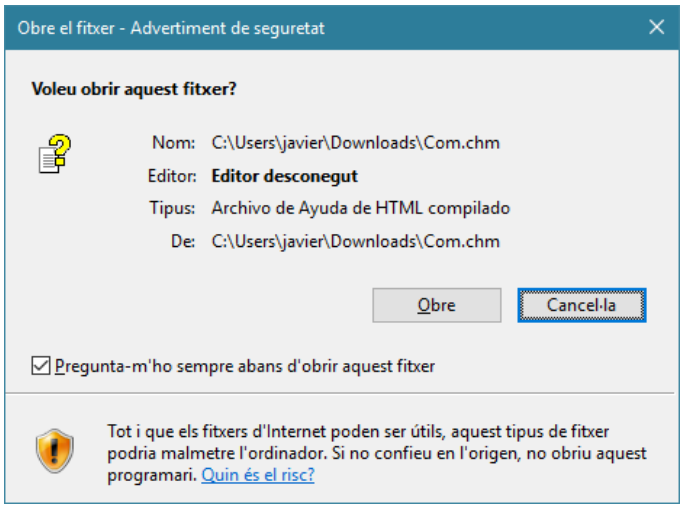


Dentro de esta pantalla inicial se presentan 5 opciones diferentes:

1. Archivo: Este menú contiene las opciones básicas de creación y guardado de archivos, así como la creación de proyectos, sobre los que hablaremos más adelante. Los archivos código fuente de CoSeL tendrán la extensión '.csl' y se pueden editar tanto dentro del entorno CrossVisions como mediante cualquier editor de texto (Notepad ++ por ejemplo).
2. Editar: Funciones típicas de edición de texto (Copiar, Pegar, Cortar, Deshacer).
3. Ver: Visibilidad de las barras de herramientas y estado.
4. Ventana: Opciones de visualización en ventanas: Cascada, Mosaico, etc ...
5. Ayuda: Inicio de la ayuda de CrossVisions



La ayuda sobre CoSeL la podéis bajar del campus virtual, pero directamente no funcionará por las incompatibilidades que ha generado Windows con no permitir abrir ficheros bajados de internet. Para solucionarlo tendréis que indicar que para este fichero no vuelva a preguntar si queremos abrir el fichero (desactivar el check de abajo a la izquierda del diálogo). Si no lo hacéis la ayuda aparecerá como páginas vacías. Este mismo problema aparece con la ayuda del módulo com.



Como se ha indicado anteriormente, una buena práctica a la hora de trabajar en CoSeL es crear proyectos para cada una de las prácticas que se vayan a hacer ya que automáticamente crea un ejecutable desde el cual acceder directamente a la interfaz de comandos. El fichero '.CIP'

asociado automáticamente carga el fichero 'init.csl' si lo hubiera. En el caso de las prácticas de esta asignatura ya se ha creado un fichero “init.csl” con el siguiente contenido:

```
// Definiciones necesarias para cargar el fichero Sintactic.csl
Proc SetComponentsGrup(grup)=> {}
proc SetOpcionesInformes(VerSintactico=true, VerSemantico=true,
    VerGenCod=true,VerObligatorio=true, VerAdicional=true,
    VerCorrecto=true)=> {}

load Sintactic

Compila("LA MEVA PROVA DEL COMPILADOR",InStrStream("\{
      function factorial(n:integer):integer
      {
          if n<=0 then return 1;
          else return n*factorial(n-1);
      }
      procedure main()
      {
          print "Factorial de ",0," es ",factorial(0),"\\n";
          print "Factorial de ",5," es ",factorial(5),"\\n";
      }
\}"));

GraficArbreSintactic();
```

Gracias a esto cada vez que abramos el fichero '.CIP' automáticamente se cargará el fichero del sintáctico y se compilará la función factorial y main. Finalmente se visualiza el árbol sintáctico resultante de la compilación.

3.2. Primeros pasos

3.2.1. Comandos útiles

Cargar ficheros '.csl': load nombre_fichero_sin_extension
ejemplo: load sintactic

Declaración de variables. En este caso CoSeL no necesita que indiquemos el tipo de la variable a crear (aunque en la práctica sí que lo exigiremos) y podemos inicializar el valor de esta variable a la vez que la declaramos (cosa que la gramática de LOOS no acepta en principio):

```
var nombre_de_variable;
var nombre_de_variable=valor;
```
ejemplo:
```
var c;
var a,b,c;
var a = 20*3;
```

Podremos declarar más de una variable a la vez, pero no podremos inicializarlas a la vez con un mismo valor

error: `var a,b = 20;`
correcto: `var a = 20, b = 20;`

Declaración de cadenas de caracteres:

```
var nombre_variable = "cadena";
var c = "hola";
```

Imprimir por pantalla:

Sin salto de línea: `cout.print("cadena");`
ejemplo: cout.print("Hola")

Con salto de línea: `cout.println("cadena");`
ejemplo: cout.println("Hola");

3.2.2. Manejo de arrays y listas

Declaración de arrays:

```
var nombre_variable = new vector[tamaño_array];
```
var d = new vector[10];

Estos arrays se pueden inicializar a la vez que se declaran, como se ve en el siguiente ejemplo:

var d = (1,2);

Otra posible inicialización de vectores sería la siguiente:

var d = vector(1,2,3)

Para acceder a los elementos del array simplemente indicaremos su posición entre corchetes, teniendo en cuenta que el primer elemento comienza por el índice 0:

```
d[0] = 1;
```
d[1] = 2;

En principio los arrays funcionan como contenedores de información pudiendo mezclar tipos de datos, aunque en la práctica esto no se permitirá.

CoSeL> d = (1,"hola")

(1,"hola")

Listas. Otro tipo de dato útil es la lista de elementos. Para definir una lista de elementos se puede hacer lo siguiente:

```
CoSeL> var a = [];
[]
CoSeL> a = [1,4,5,"hola"]
```
[1,4,5,"hola"]

Los elementos de la lista se pueden acceder de la misma manera que en los arrays, como vemos en el siguiente ejemplo:

```
CoSeL> a[0]
1
CoSeL> a[3]
```
"hola"

Otra manera útil de acceder a los elementos de una lista es mediante los comandos Head and Tail. Veámoslo con un ejemplo:

```
var m = [1,2,3,4];
m.Head = 1
m.Tail = [2,3,4]
m.Tail.Head = 2
```

Como se ha visto, se pueden concatenar diferentes operaciones Head y Tail para recorrer los distintos elementos de una lista. Otra alternativa para recorrer los elementos de una lista se muestra a continuación:

```
CoSeL> for (i<-m) cout.println(i);
1
2
3
4
```

En este caso tenéis que observar que los elementos de la lista se recorren de izquierda a derecha. Para invertir el orden de la lista es útil la operación reverse:

```
CoSeL> reverse(m)
[4,3,2,1]
```

Finalmente podemos averiguar el tamaño de una lista mediante la función length. Este comando también puede ser usado para organizar un recorrido de la lista como vemos a continuacion:

```
CoSeL> for(i<-0..a.length()-1)
cout.println(a[i])
1
4
5
hola
```

3.2.3. Otras consideraciones

Por último, se enumeran en esta sección algunas consideraciones que pueden ser útiles al realizar las prácticas:

Formatear los comentarios: En CoSeL los comentarios se pueden introducir de dos maneras. Si queremos poner comentarios de una sola línea usaremos el comando "//" del siguiente modo:

```
// Esto es un comentario
```

En cambio si queremos comentar gran parte del código la mejor opción es delimitar la parte a comentar mediante el uso de "/*" y "*/".

```
/* Esto es un comentario
un poco largo que ocupa
mas de una linea */
```

Operaciones Aritiméticas. CoSeL tiene definidas las operaciones aritméticas más comunes como la suma, resta, multiplicación y división. La precedencia de operaciones es la usual, siendo preferentes multiplicaciones y divisiones a sumas y restas. Es decir, a + c * d equivale a a + (c*d):

```
CoSeL> 20 + 5 * 4
40
```

Otras operaciones. También están definidas otras operaciones como la raíz cuadrada (sqrt) o la potencia (**). Hay que tener en cuenta que salvo en operaciones de suma y resta el tipo de la variable de salida será real.

4. Análisis Sintáctico con el módulo COM

Tal y como se ha explicado en las clases de teoría el análisis sintáctico tiene como objetivo comprobar que todas las 'palabras' de las que consta nuestro sistema sean reconocidos por nuestro sistema. Por tanto, tendrá que comprobar que cada una de las instrucciones que aparecen a lo largo del código tengan su representación dentro del árbol sintáctico y, si no es el caso, el código concreto no podrá ser compilado.

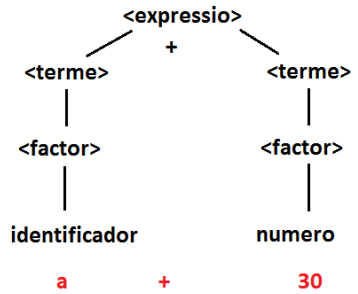
Veamos el siguiente ejemplo con una gramática más reducida que la de LOOS:

```
Use com
Var Gramatica=
BNF_Grammar <expressio>
Terminals + - * / ( ) identificador numero
BNF
      Rule <expressio>::= <terme>{(+|-) <terme>}
      Rule <terme>::= <factor>{(*|/) <factor>}
      Rule <factor>::= [-]("(" <expressio> ")" |identificador |
      numero)
End;
```

Ahora imaginemos que el código que tenemos que analizar es el siguiente:

```
a + 30;
c ** 2;
```

La primera línea puede ser reproducida perfectamente por la gramática, como se ve en la Figura

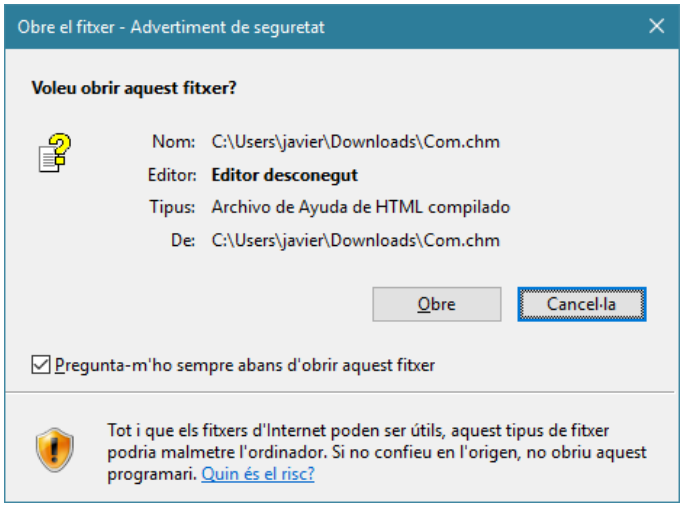


En cambio, la sentencia 'c ** 2' no puede ser reproducida mediante esta gramática al no estar incluido el doble asterisco (**) como operación permitida. Si quisiésemos que este programa se compilase tendríamos que añadir en la gramática dicha regla, tal y como haremos en la práctica. Un ejemplo de modificación (**que NO es lo que hay que hacer en esta práctica**) sería el siguiente:

```
Use com
Var Gramatica=
BNF_Grammar <expressio>
Terminals + - * / ( ) identificador numero
BNF
Rule <expressio>::=<terme>{(+|-) <terme>}
Rule <terme>::=<factor>{(*|/|**) <factor>} // No es la solucion
correcta
Rule <factor>::=[-]("(" <expressio> ")" |identificador | numero)
End;
```

Dentro del paquete software para realizar las prácticas se ha incluido la función GraficArbreSintactic que mostrará por pantalla el árbol sintáctico correspondiente al último código que se ha cargado en memoria.

La ayuda sobre el módulo com.csm se puede descargar del aula Moodle del Compiladors, pero directamente no funcionará por las incompatibilidades que ha generado Windows con no permitir abrir ficheros bajados de internet. Para solucionarlo tendréis que indicar que para este fichero no vuelva a preguntar si queremos abrir el fichero (desactivar el check de abajo a la izquierda del diálogo). Si no lo hacéis la ayuda aparecerá como páginas vacías.



5. Gramática de LOOS

Aunque la gramática de LOOS se ha explicado ya durante las clases de teoría se incluye un desglose funcional de la gramática a modo de referencia:

5.1. Símbolos terminales

En este listado están todos los símbolos terminales que permite el lenguaje LOOS. Todo terminal que no esté en esta lista no será reconocido por tanto si se ha de ampliar el lenguaje con nuevas funciones también se tendrá que modificar este listado.

```
BNF_PARSER <programa>
TERMINALS
    + - * / identificador numero caracter string ( ) =
    == != > < >= <= ^
    ! && ||
    , ; : . & "::"
    Type var Function Procedure constructor destructor
    Integer Real Character bool class array of "[" "]" "{" "}"
    if then else while do return
    Print Exit
    delete new
    true false null this operator
    IMPRIMIR_TAULA_DE_SIMBOLS
```

5.2. Reglas BNF

Cualquier línea de código del programa deberá ser reproducida mediante las reglas BNF que se exponen a continuación:

```
BNF
Rule <programa>::=
    {
        <DecFun>
        | <DecProc>
        | <DecConstructor>
        | <DecDestructor>
        | <DecVar>
        | <DecTipus>
        | <DecClasse>
    }
```

5.2.1. Tipos de datos

```
Rule <DecTipus>::=
    "Type" identificador = <tipus> ";"

Rule <tipus>::=
        Integer
        | Real
        | Character
        | Bool
        | ^ <tipus>
        | Array "[" numero "]" of <tipus>
 | identificador [<Parametres>]
```

5.2.2. Definición de clases y sus elementos

Las clases en LOOS constan de:

- Atributos
- Métodos
- Constructores
- Destructores
- Asignador

Se recuerda que en el cuerpo de la declaración de la clase sólo se incluirán los prototipos de los métodos y las funciones constructoras y destructoras. El cuerpo de la función deberá ser definido después teniendo en cuenta la notación específica de un lenguaje orientado a objetos, como se ve en el ejemplo:

```
Class nodo
{
    valor_x : integer;
    valor_y : integer;
    Constructor nodo(x:integer,y:integer);
    Destuctor nodo();
    function nodosuma():integer;
}

Constructor nodo(x:integer,y:integer)
{
    valor_x = x;
    valor_y = y;
}

function nodo::nodosuma()
{
    var resultado;
    resultado = this^.valor_x + this^.valor_y;
    return resultado;
}

procedure main()
{
    var n1;
    var res;
    n1:nodo(20,30);
    res = n1.nodosuma();
}
```

La gramática correspondiente a las clases es:

```
Rule <DecClasse>::=
  "Class" Identificador [":" Identificador ] "{" {<ClassElement>} "}"

Rule <ClassElement>::=
    identificador : <tipus> ";" |
    <ProcPrototipus> ";" |
    <FunPrototipus>  ";" |
```

```
    <ConstructorPrototipus> ";" |
    <DestructorPrototipus>  ";"

Rule <FunPrototipus>::=
    function identificador ["::" identificador] <DecParams> : <Tipus>

Rule <ProcPrototipus>::=
    procedure identificador ["::" identificador] <DecParams>

Rule <ConstructorPrototipus>::=
    Constructor identificador <DecParams>

Rule <DestructorPrototipus>::=
    Destructor identificador <DecParams>

Rule <DecParams>::=
    "(" [ <DecParametre> { , <DecParametre> } ] ")"

Rule <DecParametre>::=
     identificador : <tipus>
```

5.2.3. Declaraciones

```
Rule <DecVar>::=
      Var identificador : <tipus> ";"

Rule <DecFun>::= <FunPrototipus> <bloc>

Rule <DecProc>::= <ProcPrototipus> <bloc>

Rule <DecConstructor>::= <ConstructorPrototipus> <bloc>

Rule <DecDestructor>::= <DestructorPrototipus> <bloc>
```

5.2.4. Bloques de instrucciones del programa

```
 Rule <bloc>::=
    "{"
    {
        <DecFun>
        | <DecProc>
        | <DecVar>
        | <instruccio>
     }
     "}"

 Rule <instruccio>::=
     <Expressio> ";"
     | return [ <Expressio> ] ";"
     | if <Expressio> then <instruccio> [else <instruccio> ]
     | while <Expressio> do <instruccio>
     | <bloc>
```

```
| ";"
| Print <Expressio> { ,<Expressio> } ";"
| Exit ";"
| delete <expressio>
```

5.2.5. Expresiones

```
Rule <Expressio>::=
        <TerBool> { || <TerBool> }

Rule <TerBool>::=
        <FacBool> { && <FacBool> }

Rule <FacBool>::=
        <ExpArit> [ ( == | != | ">" | "<" | >= | <= ) <ExpArit> ]

Rule <ExpArit>::=
        <terme> { ( + | - ) <terme> }

Rule <terme>::=
    <factor> { ( * | / ) <factor> }

Rule <factor>::=
      ! <factor>
    | - <factor>
    | & <factor>
    | "(" <Expressio> ")"
    | Numero
    | Caracter
    | String
    | true
    | false
    | Null
    | "new" <tipus> [<Parametres>]
    | this <PostFixe>
    | Identificador [<Parametres>] <PostFixe>

Rule <Postfixe>::=
    <Acces> [ = <Expressio> ]

Rule <Acces>::=
    {
          "[" <Expressio>"]"
        | . identificador [<Parametres>]
        | ^
    }

Rule <parametres>::="(" [ <Expressio> { , <Expressio> } ] ")"
END;
```

6. LOOS+

Una vez presentada la gramática del Lenguaje Orientado a Objetos Simple (LOOS) la propuesta es añadir nuevas funcionalidades a esta gramática. Como ya se ha explicado en la presentación de la asignatura, este año cada grupo tendrá una práctica diferente, aunque la metodología para resolverla es común a todas. Hemos agrupado las diferentes ampliaciones en grupos según las funcionalidades que añaden. Es muy importante que no olvidéis el objetivo del Análisis Sintáctico, que es que las diferentes instrucciones que os proponemos puedan ser interpretadas por el compilador, independientemente del resultado que den (sobre todo en el caso de operadores matemáticos); ya nos preocuparemos en prácticas posteriores de temas como comprobaciones de tipos de datos o si las variables existen o no (Análisis Semántico) o de obtener la salida que tendría que dar el código (Generación de Código).

Presentamos a continuación todas las ampliaciones propuestas en este curso. Cada alumno tendréis asignada un subconjunto de todas las ampliaciones. Podéis identificarlas a partir del código que va entre corchetes seguido del título de cada ampliación. Como la práctica se entrega en grupos de dos alumnos, podéis seleccionar cuál de los dos subconjuntos de las ampliaciones queréis entregar. Las ampliaciones a entregar serán las del alumno que vaya primero en la identificación de la práctica que rellenáis en el fichero CoSeL (.csl) que entreguéis.

6.1. Declaraciones de variables

Este conjunto de ampliaciones se basa en ampliar las funcionalidades de LOOS en cuanto a la declaración de varios elementos del mismo tipo, ya sea variables o parámetros en una función, por ejemplo. La idea es permitir al futuro programador reducir el código que se necesita programar manteniendo la funcionalidad.

6.1.1. Declaración de múltiples variables con el mismo tipo de datos [DeclaracionMultiplesVariablesMismoTipo]

En la implementación original de LOOS si queremos declarar dos variables del mismo tipo hemos de hacerlo en dos líneas de código diferentes. Se propone que la gramática de LOOS+ permita **declarar varias variables de un mismo tipo a la vez**, del estilo:

```
var a, b, c:integer;
var d,e:real;
var obj1,obj2:Class1(10,20);
```

Si se combinara esta ampliación con la ampliación de DeclaracionMultiplesVariablesDiferenteTipo, será posible hacer declaraciones con el siguiente aspecto:

```
var a, b, c:integer, d:real;
var d,e:real, tabla1,tabla2:array [20] of integer;
```

A nivel semántico habrá que considerar que no se declaren variables con el mismo nombre en el mismo ámbito.

A nivel de generación de código se tendrá que considerar que cada variable ha de tener un espacio de memoria diferente y que, en el caso de inicializar las variables mediante un constructor, habrá que llamar al constructor para cada una de las variables. Ejemplo:

```
Var a,b:MyClass(x*2);
```

Declara las variables a y b de clase MyClass. Se llamará dos veces al constructor de MyClass, una para la variable a y otra para la variable b.

6.1.1.1. Análisis Sintáctica

Hay que modificar la regla <DecVar> para añadir la sintaxis "var a,b,c:integer;" que permite declarar varias variables del mismo tipo, cuando LOOS solo soportaba "var a:integer;" que sólo permite declarar una variable.

6.1.1.2. Análisis Semántica

Al añadir la sintaxis que permite declarar varias variables con el mismo tipo de datos hay que crear una entrada de variable para cada una de las variables:

Por ejemplo, la declaración "var a,b,c:integer;" supone añadir a la tabla de símbolos las entradas: EtsVariable("a",TInt), EtsVariable("b",TInt) y EtsVariable("c",TInt). El problema es que no sabemos el tipo de las variables hasta que llegamos al final de la declaración donde aparece integer. Por otra parte hay que detectar si estas variables están duplicadas con otras variables que hay en la tabla de símbolos o con ellas mismas (var a,a:integer).

Una forma de hacer esto es verificar las declaraciones duplicadas en la tabla de símbolos con TS.ComprovarDuplicat(nombre). Esto obliga a que cada vez que aparece un nombre de una variable ya crear su ETSVariable e insertarla en la tabla de símbolos aunque no tengamos el tipo de datos. Así cuando aparece la variable "a" creamos la entrada de la tabla de símbolos ETSVariable("a",unbound), la guardamos en una lista de entradas y la insertamos en la tabla de símbolos con TS.Insertar(entrada). Luego cuando tengamos el tipo de datos podemos recorrer la lista de entradas y a cada una de ellas le asignamos el tipo de datos que nos da el símbolo no terminal <tipus(t)> (ets.Tipus=t).

6.1.2. Declaración de múltiples variables con diferente tipo de datos [DeclaracionMultiplesVariablesDiferenteTipo]

En la implementación original de LOOS si queremos declarar dos variables de diferente tipo hemos de hacerlo en dos líneas de código diferentes. Se propone que la gramática de LOOS+ permita **declarar varias variables de diferente tipo a la vez**, del estilo:

```
var a:Real, b:Integer, c:integer;
var obj1:class2(5,6), obj2:class2(9);
var t:array [5] of real, x:integer;
```

A nivel semántico habrá que considerar que no se declaren variables con el mismo nombre en el mismo ámbito.

A nivel de generación de código se tendrá que considerar que cada variable ha de tener un espacio de memoria diferente. También hay que considerar que se ha de llamar a constructor si se está declarando un objeto.

6.1.2.1. Análisis Sintáctica

La sintaxis de declaración de variables en LOOS sólo permite declarar una variable. En esta ampliación permitirá declarar múltiples variables en una sentencia "var". Así podemos considerar que sintácticamente una declaración de multiples variables corresponde a una sentencia de declaración de una variable a la que le quitamos el "var" y el ";" del final y repetimos lo que queda separado por comas y al final añadir el "var" al principio y el ";" al final. Por ejemplo, si queremos definir la sintaxis "var a:integer,b:real,c:character;", la podemos construir considerando que teníamos tres declaraciones "var a:integer;", "var b:real;" y "var c:character;".

Quitamos los “var” y “;” y obtenemos “a:integer”, “b:real” y “c:character". Luego los juntamos poniendo comas entre ellos “a:integer, b:real, c:character” y finalmente añadimos “var” y “;”. El resultado es la declaración de múltiples variables “var a:integer, b:real, c:character;”. Por lo tanto, lo que nos interesa es crear una regla <DecUnaVar> que solo genera la declaración de variable sin “var” ni “;” y luego cambiar <decVar> de forma que sea “var” más la repetición de una o más veces de <DecUnaVar> separada por comas y finalmente “;”. De esta forma tendremos que todas las acciones semánticas y de generación de código que tengamos que añadir posteriormente, solo habrá que ponerlas una vez en la regla de <DecUnaVar>.

6.1.2.2. Análisis Semántica

En esta ampliación, las acciones semánticas son las mismas que se hacen en <DecVar>, pero trasladadas a la regla de <DecUnaVar>. La cuestión será que estas acciones semánticas cambiarán si se combina esta ampliación con otras formas de declarar variables. En todo caso estas nuevas formas de declaración de variables solo afectaran a la regla <DecUnaVar>.

6.1.3. Declaración de variables con selección automática del tipo de datos [DeclaracionVariableAuto]

En la implementación original de LOOS si queremos declarar una variable e inicializarla hay que hacerlo en dos líneas de código diferentes. Además, sabiendo el tipo de datos del valor con que inicializamos una variable nos podremos ahorrar indicar su tipo de datos. Se propone que la gramática de LOOS+ permita **declarar una variable inicializada y que el compilador seleccione automáticamente su tipo de datos**, del estilo:

```
var a=10; // Variable a de tipo integer
var b=2*3.14; // variable b de tipo real
var c=class1(5,a);
var d=c;
var s="Hola"; // variable de tipo array [4] of character
```

Si se combina esta ampliación con [DeclaracionMultiplesVariablesDiferenteTipo] se podrá escribir:

```
var a=10, b=2*3.14;
var a, b=10; // ERROR a no se inicializa y su tipo no está definido!
```

A nivel semántico se tendrá que detectar la declaración de variables duplicadas. En el caso que el valor con que se inicializa la variable se un objeto, habrá que ver si se puede llamar a su constructor por copia para inicializar la variable. Ejemplo:

```
Var obj1:Class1(20); // Construcción de obj1
Var obj2=obj1; // Construcción de obj2 llamando al constructor
               // por copia de Class1 con parámetro obj1.
               // obj2 es de tipo Class1
```

La inicialización de las variables auto no es una asignación, si no que se trata de una copia de un valor, por lo tanto, se ha de utilizar el constructor por copia y no el asignador.

6.1.3.1. Análisis Sintáctica

La declaración de variables auto afecta a la regla <DecVar>. Ahora esta regla tendrá que admitir declaraciones en la forma “var a:integer;” y en la forma “var a=10;”. Estas declaraciones solo cambian después del identificador, así que solo hay que añadir una unión con dos casos. El

primero será el que ya tenia la regla de LOOS y el segundo será el caso del igual correspondiente a la declaración de variable auto.

6.1.3.2. Análisis Semántica

Hay que mantener las acciones semánticas que ya se hacían para en <DecVar>. En el caso de declaración de variable auto habrá que hacer la comprobación de declaración duplicada, crear la entrada de la tabla de símbolos ETSVariable pero el tipo de datos vendrá del resultado de la expresión y no de <Tipus>. Finalmente habrá que insertar la entrada en la tabla de símbolos. Hay que añadir otra acción semántica que se produce porque hay que copiar del resultado de la expresión a la variable. Habrá que comprobar que realmente se puede hacer la copia (ComprovarConstruiblePerCopia).

6.1.4. Declaración múltiples elementos de una clase [DeclaracionMultiplesCampos]

En la creación de una clase se ha de incluir una línea para cada atributo independientemente si se van a declarar varios del mismo tipo de datos. En este sentido proponemos que, cuando se cree una clase, se ha de permitir **crear diversos elementos de la misma clase con el mismo tipo**, siguiendo el siguiente ejemplo:

```
Class nodo
{
    valor_x, valor_y: integer;
    Constructor nodo(x:integer,y:integer);
}
Constructor nodo(x:integer,y:integer)
{
    valor_x = x;
    valor_y = y
}
```

6.1.5. Declaración múltiples parámetros del mismo tipo al declarar una función [DeclaracionMultiplesParametros]

En la declaración de una función se han de incluir de manera ordenada los parámetros que ésta recibe. En la implementación original se han de declarar los parámetros uno a uno, incluso si tenemos varios del mismo tipo consecutivos. Proponemos la siguiente ampliación: cuando estemos declarando una función, hemos de permitir indicarle la **creación de varios parámetros del mismo tipo a la vez**:

```
function f1(a,b:integer,c:real):integer ...
procedure p1(a,&b:Real) ... // a se pasa por valor y b por referencia
```

A nivel semántico hay que considerar que no duplique el nombre de los parámetros y tratar correctamente cuando se pasan por referencia y cuando no. El tipo de datos podrá ser compartido por varios parámetros.

6.1.6. Declaración de parámetros de funciones de entrada, salida y entrada-salida [DeclaracionParametrosInOut]

En LOOS se pueden declarar parámetros de funciones pasados por valor y por referencia. En esta ampliación del lenguaje LOOS queremos ampliar los modos de paso de parámetros a parámetros de entrada, parámetros de salida y parámetros de entrada-salida. Por lo tanto los modos de paso de parámetros que tendremos después de la ampliación serán:

- Paso por valor: `parámetro:tipo`. Se copia el valor del parámetro antes de entrar en la función. Dentro de la función se puede leer y modificar el valor del parámetro. Esto solo afecta a la copia del valor del parámetro que se ha hecho al llamar a la función.
- Paso por referencia: `&parámetro:tipo`. Se copia la dirección del parámetro antes de entrar en la función. Dentro de la función se puede leer y modificar el valor del parámetro. Esto afecta a la variable (left-value) que se ha pasado como parámetro.
- Parámetro de entrada: `in parámetro:tipo`. Se copia el valor del parámetro antes de entrar en la función. Dentro de la función se puede leer, pero no modificar el valor del parámetro.
- Parámetro de salida: `out parámetro:tipo`. Antes de llamar a la función se construye el parámetro con su constructor por defecto si es necesario. Luego dentro la función solo se puede asignar valores al parámetro. No se puede leer el valor del parámetro. Al salir de la función se asigna el valor del parámetro en la variable (valor con permiso de escritura) que se pasó como parámetro.
- Parámetro de entrada-salida: `in out parámetro:tipo`. Se copia el valor del parámetro antes de entrar en la función. Dentro de la función se puede leer y modificar el valor del parámetro (copia local). Al salir de la función se asigna el valor del parámetro en la variable (valor con permiso de lectura y escritura) que se pasó como parámetro.

No se permiten otros modos de paso de parámetros que los indicados anteriormente. Por lo tanto, no se pueden combinar el paso de parámetros por referencia con los parámetros de entrada o salida.

Los parámetros de entrada/salida se pueden utilizar en las declaraciones y definiciones de funciones, procedimientos, constructores, asignadores y métodos.

Trabajar con parámetros de entrada/salida supone que se tendrá que considerar para las variables y valores permisos de lectura y escritura. Esto pasa porque un parámetro de una función es una variable local de la función y porque si una variable tiene unos permisos, estos los heredan cualquier elemento del tipo de datos de la variable. Por ejemplo, en la función: `function f(in a:array [10] of integer):integer`, a es un parámetro de entrada, por lo tanto la variable local a solo tiene permiso de lectura. Con a tiene el permiso de solo lectura, los elementos de a (a[i]) heredan este permiso de solo lectura. Así la expresión `a[1]=5` tendría que generar el error de que no se puede modificar el valor de a[1].

Ejemplos de declaraciones correctas:

```
Function f(a:integer, &b:real, in c:integer, out d:bool, in out
e:array [3] of real)…
Procedure p(out d:integer)…
Constructor p(out d:integer,in x:real, in out y:real)…
Procedure myClass::Metodo(in d:integer)…
```

Ejemplos de declaraciones incorrectas:

```
Function f(in &c:integer, out &d:bool, in out &e:array [3] of real)…
Function f(out in e:array [3] of real)…
```

Esta ampliación se puede combinar con las ampliaciones [DeclaracionMultiplesParametros] de la misma forma que se combina con el paso de parámetros por referencia. Así cada parámetro de un grupo con el mismo tipo puede tener un paso de parámetros diferente. Ejemplo:

Function f(in a1, out a2, int out a3, &a4, a5:integer, &b:real)…

Los parámetros a1, a2, a3, a5 y a5 son de tipo entero. El parámetro a1 es de entrada, el parámetro a2 es de salida, el parámetro a3 es de entrada-salida, el parámetro a4 se pasa por referencia y el parámetro a5 se pasa por valor. El parámetro b es de tipo real y se pasa por referencia.

6.1.6.1. Análisis Sintáctica

En esta ampliación hay que añadir los modificadores in, out e in out a las declaraciones de los parámetros. Esta modificación se ha de hacer en la regla <DecParametre> Habrá que hacer una unión con tres ramas, una para el paso por referencia "&", otra para "in" que opcionalmente puede seguirle "out" y una última para "out" sin "in".

6.1.6.2. Análisis Semántica

En el análisis de semántico de cada parámetro en <DecParametre> hay que crear una estructura Parametre que se ha de añadir a la lista de parámetros de la regla <DecParametres>. La estructura Parametre contiene los siguientes campos:

```
Type [public,Print=Contents] Parametre(
    Nom, // Nom del paràmetre
    Tipus, // Tipus del paràmetre
    Referencia, // Pas per referencia=true per valor=false
    ParametreIn, // Paràmetre d'entrada (es pot llegir)
    ParametreOut // Paràmetre de sortida (es pot escrire)
    );
```

Así tendremos según el modo de paso del parámetro los siguientes casos:

- Paso por valor (a:integer): Referencia= false, ParametreIn= false, ParametreOut= false.
- Paso por referencia (&a:integer): Referencia=true, ParametreIn= false, ParametreOut= false.
- Parámetro de entrada (in a:integer): Referencia= false, ParametreIn= true, ParametreOut= false.
- Parámetro de salida (out a:integer): Referencia= false, ParametreIn= false, ParametreOut= true.
- Parámetro de entrada y salida (in out a:integer): Referencia= false, ParametreIn= true, ParametreOut= true.

La cuestión es que para cada modo de paso del parámetro han que hacer unas operaciones con el valor, y hay que comprobar que esas operaciones son posibles. Esto supone añadir las comprobaciones de si se puede construir el parámetro por copia (ComprovarConstruiblePerCopia) o si se puede asignar (ComprovarAssignable) o construir por defecto (ComprovarConstruiblePerDefecte).

Las comprobaciones por hacer según el modo de paso del parámetro serán:

- Paso por valor (a:integer): Construible por copia.
- Paso por referencia (&a:integer): Ninguna.
- Parámetro de entrada (in a:integer): Construible por copia.
- Parámetro de salida (out a:integer): Construible por defecto y asignable.
- Parámetro de entrada y salida (in out a:integer): Construible por copia y asignable.

Con lo anterior hemos tratado de que pasa cuando se añaden los parámetros de entrada y/o salida en las declaraciones de parámetros en los prototipos de clausuras. Ahora hay que considerar al definir una clausura (prototipo + cuerpo) cómo se convierten estos parámetros en variables locales de la clausura. Hay que considerar que a los parámetros de entrada les corresponden variables que solo se puede leer su contenido y no se pueden modificar. A los parámetros de salida les corresponde variables que se puede modificar su contenido, pero no se puede leer. Las entradas de variable ya están preparadas para estos casos con campos que indican si tenemos permiso de lectura y escritura de la variable. La estructura de una entrada de la tabla de símbolos de variable es la siguiente:

```
Type [public,Print=Contents] ETSVariable(
        Tipus, // tipus de la variable
        PermisLectura, // Es pot llegir el valor de la variable
        PermisEscriptura, // Es pot escriura la variable
        …
    ) from EntradaTaulaSimbols
```

Al crear las variables locales de la clausura habrá que decidir qué valores han de tener los campos PermisLectura y PermisEscritura según el modo de paso de cada parámetro. Los casos son los siguientes:

- Paso por valor (a:integer): PermisLectura= true, PermisEscritura= true.
- Paso por referencia (&a:integer): PermisLectura= true, PermisEscritura= true.
- Parámetro de entrada (in a:integer): PermisLectura= true, PermisEscritura= false.
- Parámetro de salida (out a:integer): PermisLectura= false, PermisEscritura= true.
- Parámetro de entrada y salida (in out a:integer): PermisLectura= true, PermisEscritura= true.

Esta creación de los variables locales a partir de los parámetros pasara en las reglas de definición de clausuras (<DecFun>, <DecProc> y <DecConstructor>).

Una vez tratada la declaración de parámetros, falta modificar las llamadas a clausuras. Estas se producen en las reglas <PosibleLValue> que trata casos como “f(x,y)” y < AccesCampMetode> que trata llamadas a métodos “obj.m(x,y)”. Estas dos funciones llaman a la regla <Parametres> para obtener la lista de descriptores de los parámetros, luego llaman a SeleccionarMetode para seleccionar la sobrecarga que se aplicará y finalmente a ComprovarPermisosParametres. Esta última es la que controla si los permisos del valor que se pasa como parámetro son adecuados para el modo de paso del parámetro. Estos permisos han de coincidir con los permisos indicados para las variables locales creadas a partir de los parámetros. Hay que considerar que la estructura de descriptor de valor ya soporta los permisos de lectura y escritura:

```
Type [public] DescriptorValor(
    Tipus,           // Tipus de dades
    PermisLectura,   // Es pot llegir el contingut d'aquet valor.
    PermisEscriptura, // Es pot escriure en el contingut d'aquet valor
    …
    );
```

Finalmente, un efecto que supone la existencia de parámetros de salida es que crean variables que solo tienen permiso de escritura. Esto supone que existirán valores sólo con permiso de escritura. Por lo tanto, en todo lugar del compilador donde se quiera leer un valor habrá que

comprobar que tiene permiso de lectura. Lo mismo pasa con los parámetros de entrada que solo tienen permiso de lectura. En este caso, utilizar un parámetro de entrada como índice de un bucle for tendría que dar error ya que no se puede modificar para incrementarlo.

6.2. Manejo de Constantes

6.2.1. Declaración de constantes [DeclaracionConstante]

LOOS+ permitirá declarar variables con valor constante. Sólo permitiremos la declaración de constantes de tipo entero, real, booleano o carácter. Podemos ver a continuación un ejemplo de la declaración de constantes en LOOS+:

```
const a=1;
const b=2.2;
const c='a';
const d=true;
```

En el caso que además de esta ampliación se implemente la ampliación de AritmeticaConstante, será posible definir el valor de una constante con una expresión aritmética:

```
const a=1+3;
const b=2*a;
```

También se pueden definir constantes miembro de una clase. Simplemente se han de añadir dentro de las llaves de la clase de la siguiente forma:

```
class MyClass {
    a=10;
    b=’x’;
}
```

Si tenemos que implementar la ampliación de AritmeticaConstante se podrá inicializar las constantes con expresiones aritméticas cuyo resultado sea una constante. Ejemplo:

```
class MyClass2 {
    a=10*4;
    b=3+4*2;
    c=a+b;
}
```

A nivel semántico habrá que considerar que no se duplique las declaraciones de constantes. También será importante asegurar que la expresión con que se define la constante dé como resultado una constante. Además, a nivel semántico ya se tratará con el valor de la constante.

6.2.2. Aritmética de constantes [AritmeticaConstante]

Además de permitir la declaración de variables con valor constante, LOOS+ incorporará funcionalidades que permitirán operar directamente con constantes, posibilitando ahorro de instrucciones y espacio de memoria ocupado por el programa. El compilador deberá calcular los tipos de los datos de las operaciones aritméticas y tratar de manera especial aquellos casos en que ambos operandos sean constantes (sólo para operaciones de suma, resta, multiplicación, producto y cambio de signo). Podemos ver a continuación un ejemplo de la aritmética de constantes en LOOS+:

```
var a:Integer;
```

```
var b:Integer;
a=30;
b=40;
printLn "a+b=",a+b;
printLn "50+b=",50+b;
printLn "a+60=",a+60;
PrintLn "10+20=",10+20; // Se calcula en tiempo de compilación
```

El código anterior sólo genera tres instrucciones de suma y la suma de 10+20 se realiza en tiempo de compilación.

A nivel semántico ya se tendrá que considerar el resultado de las expresiones constantes, ya que este se utilizará para inicializar constantes.

6.3. Manejo de Arrays

Además de variables simples, permitimos la declaración de arrays o vectores. En la versión inicial de LOOS permitimos la declaración de arrays unidimensionales pero, si se encadena su declaración (array of array of) podemos conseguir arrays multidimensionales. Proponemos dos ampliaciones: una que permita una declaración más inmediata de arrays multidimensionales y otra que permite declarar arrays por rangos.

6.3.1. Creación de Arrays Multidimensionales [ArrayMultidimensional]

En este caso permitimos la creación de arrays multidimensionales, separando las dimensiones por comas, tal y como se ve en el siguiente ejemplo:

```
var a:array[2,4] of integer;
var b:array [3,4,6] of real;
```

La variable creada es equivalente a haber hecho

```
var a:array[2] of array [4] of integer;
var b:array [3] of array [4] of array [6] of real;
```

Ya que se ha creado un array de varias dimensiones se ha de considerar acceder a este utilizando directamente un acceso por varios índices

```
var a:array[2] of array [4] of integer;
var b:array[2,4] of integer;
var c:array[5,3,10] of real;
a[i,j]=10;
a[i][j]=10;
b[i+2,8]=b[i+3,j*15];
c[i+1][j+3,k+8]= 5 * c[i+2,j+1][k*2];
```

A nivel semántico y de generación de código se tendrá que considerar que:

- array [i_1,i_2,..,i_n] of … es equivalente a array [i_1] of array [i_2] of … array [in] of …
- identificador[i_1,i_2,i_3,..,i_n] es equivalente a identificador[i_1][i_2][i_3]…[i_n].

6.3.1.1. Análisis sintáctico

En esta ampliación hay que modificar la sintaxis de los tipos de datos, por lo tanto, hay que modificar la regla <tipus>. De esta regla hay que modificar la parte donde se definen los arrays para que admita poner más de un índice entre los corchetes. Esto permitirá definir arrays con la sintaxis “array [10,20,40] of”

Como ya se pueden definir arrays de varias dimensiones habrá que modificar el acceso por índice para que acepte la sintaxis “tabla[i,j,k]” con múltiples índices cuando LOOS solo aceptaba la sintaxis “tabla[i]” con un solo índice que se encuentra en la regla <AccesIndex>.

6.3.1.2. Análisis semántico

En la sintaxis “array [10,20,30] of integer” que permite definir arrays de varias dimensiones en la regla <tipus>, hay que añadir las acciones semánticas necesarias para que cree el correspondiente descriptor de tipos. En este caso será TArray(10, TArray(20, TArray(30, TInt))).

El problema que aparece para crear la estructura TArray(10, TArray(20, TArray(30, TInt))) es que justamente tenemos ir haciendo crecer la estructura TArray(Mida,TipusElements) por TipusElements. Esto es debido a que primero tenemos el tamaño del array más externo (TArray(10,…)), luego del siguiente más interno (TArray(20,…)) y al final de todo tendremos el tamaño del más interno de todos y el tipo de los elementos que contienen (TArray(30,TInt)). Una idea para hacer esto es utilizar un apuntador al campo. Si queremos construir la estructura en la variable t y utilizamos p como apuntador a donde pondremos la parte del tipo de datos que queremos construir. Empezamos con p=&t, al analizar la dimensión 10 haremos *p=TArray(10,unbound) y actualizaremos p para que apunte al campo TipusElements, p=&p->TipusElements. Así cuando hagamos *p=TArray(20,unbound) sustituiremos el unbound de TArray(10,unbound) por TArray(20,unbound). Por lo tanto, t valdrá TArray(10, TArray(20, unbound)). Al final asignaremos a *p el tipo de datos de los elementos del array.

Otra forma de crear la estructura TArray(10, TArray(20, TArray(30, TInt))) es guardar las dimensiones en una lista en orden inverso. Sea Dims la lista de dimensiones entonces empezamos con dims=[] y por cada dimensión haremos dims=10::dims. El resultado será la lista dims=[30,20,10]. Como las dimensiones están a la inversa podremos construir la estructura TArray(10, TArray(20, TArray(30, TInt))) de dentro hacia fuera. Empezamos con t igual al tipo de los elementos del array t=TInt, Luego queremos añadir la primera dimensión de la lista. Entonces hacemos t=TArray(30,t), el resultado será t=TArray(30,TInt). Si repetimos con el siguiente elemento de la lista haremos t=TArray(20,t) y el resultado será t=TArray(20,TArray(10,TInt)). Así lo haremos con todas las dimensiones que tengamos guardadas en la lista.

En la regla < AccesIndex> para una dimensión ya comprueba que la expresión que calcula el valor del índice sea de tipo TInt y cambia el tipo array por el tipo de los elementos del array. Esto se hace con el código:

```
if (dvIndex.Tipus!:=TInt) throw SemanticError("Tipus erroni en index");
dvAcc.Tipus=dvAcc.Tipus.TipusElements;
```

Así que para hacerlo con el resto de las dimensiones que podemos encontrar entre los corchetes, habrá que repetir lo mismo, aunque habrá que tener en cuenta si el tipo de datos sea TArray cuando corresponda en aquellos casos en que vamos navegando por las diferentes dimensiones.

6.3.2. Creación de Arrays con rangos de posiciones [ArrayRangos]

En esta ampliación definiremos una serie de posiciones en las que el array puede tener valores, siguiendo la sintaxis siguiente:

```
var a:array[2..4] of integer;
var b:array[-2..4] of real;
```

En este caso, la variable a creada es un array unidimensional donde sólo tenemos definidas las posiciones 2, 3 y 4. La variable b tiene posiciones -2, -1, 0, 1, 2, 3, 4. Así a[2] es el primer elemento del array a.

En un array definido por un rango [v1..v2], el primer valor ha de ser menor o igual que el segundo.

6.3.2.1. Análisis sintáctico

En esta ampliación hay que modificar la sintaxis de los tipos de datos, por lo tanto, hay que modificar la regla <tipus>. De esta regla hay que modificar la parte donde se definen los arrays para que admita poner rangos además de números entre los corchetes. Esto permitirá definir arrays con la sintaxis "array [-5..30] of".

El acceso a los arrays sigue siendo con un solo índice en la regla <AccesIndex>, así que a nivel sintáctico no hay que tocar nada.

6.3.2.2. Análisis semántico

Cuando tenemos un array con rangos, hay que considerar como será su descriptor del tipo de datos. En este caso utilizaremos la estructura TArray(Mida,TipusElements), pero en el campo Mida guardaremos un vector con el rango. Así tendremos que "array [10..20] of real" quedará representado como TArray(Vector(10,20),TReal). Por cierto Vector(10,20) se puede escribir como (10,20). En el caso que el array se defina con su tamaño tendremos en Mida un entero. Por ejemplo, "array [100] of integer" se representa como TArray(100,TInt). Hay que acordarse de que tantos los tamaños, como los límites de un rango han de ser enteros. En el caso de los tamaños estos enteros han de ser números positivos. En el caso de los rangos se la de cumplir que el límite inferior sea menor que el límite superior, pero pueden ser números negativos.

El acceso a los arrays sigue siendo con un solo índice en la regla <AccesIndex> y al acceder al array se obtiene el tipo de los elementos del array tanto si este está definido con su tamaño o con un rango, así que a nivel semántico no hay que tocar nada.

6.4. Inicializaciones de variables

Este conjunto de ampliaciones busca incrementar las posibilidades de inicialización de variables ya presentes en LOOS.

6.4.1. Inicialización de variables simples a la vez que la declaración [VarInitSimple]

En el caso de las variables simples permitiremos la inicialización de las mismas a la vez que se declaran, tal y como se ve en el siguiente ejemplo:

```
var a:integer = 20;
var b:Real=f(x,y)+15;
```

En el caso de que esta ampliación se combine con la ampliación [DeclaracionMultiplesVariablesMismoTipo] se podrán escribir declaraciones como:

```
var a,b:integer = 20; // inicializa a y b con valor 20
var c,d,e:Real=f(x,y)+15; // inicializa c y d,e con f(x,y)+15
```

En el caso de que esta ampliación se combine con la ampliación [DeclaracionMultiplesVariablesDiferenteTipo] se podrán escribir declaraciones como:

```
var a:Real=3.14,b:integer = 20;
```

y si las dos ampliaciones anteriores se combinan podremos escribir

```
var a,b,c:Real=3.14, d,e:integer = 20;
```

En el caso de inicializar objetos de clases, hay que considerar que para inicializar la variable puede ser necesario llamar al constructor por copia de la clase. También, hay que considerar que se puede inicializar un objeto de una clase A con un objeto de una clase derivada de A.

6.4.1.1. Análisis Sintáctica

Hay que modificar la regla <DecVar> para admita la sintaxis "var a:integer=10*20" y mantener la sintaxis "var a:integer(10*20)" donde se llama al constructor por copia. Estas dos sintaxis empiezan a diferenciarse cuando aparece "=" o cuando aparecen los parámetros del constructor. Por tanto, hay que hacer una unión de estos dos casos.

6.4.1.2. Análisis Semántica

El compilador de LOOS ya tiene todas las acciones semánticas para analizar "var a:integer" y le faltan las que permiten inicializar la variable con el resultado de una expresión "var a:integer=b+2". En este caso habrá que comprobar que el tipo de datos de la expresión esté incluido en el tipo de datos de la variable (TipusPertany). También se tiene que comprobar que el tipo de datos de la variable se puede construir por copia (ComprovarConstruiblePerCopia).

6.4.2. Inicialización de arrays en la declaración [VarInitArray]

Para el caso de los arrays, también contemplamos la inicialización de arrays en variables, como en el siguiente ejemplo

```
var a:array [3] of integer = {1,2,3+j};
var a:array [3] of array [2] of integer={{1,g(2)}, {4,6}, {3+j,8}};
```

6.4.3. Inicialización de objetos en declaración de variables [VarInitObjeto]

Con respecto a las clases, permitiremos la inicialización de objetos en variables, tal y como se muestra a continuación:

```
class MyClass
{
    m_a:Integer;
    m_b: real;
}

class MyClass2 : MyClass
{
    m_a2: character;
    m_b2: real;
    m_cl:MyClass;
}

procedure main()
{
   var obj1:MyClass= { 10, 30.5+x };
   var obj2:MyClass2= {{ 10, 30.5+x }, 'z', 54.1, { 10+i, 2*30.5 }};
}
```

Los valores entre llaves se utilizarán para inicializar los campos del objeto sin no tiene constructor. En el caso que la clase tenga constuctores, los valores entre llaves son los parámetros del constructor del objeto. Ejemplo con constructor:

```
class MyClass
{
    constructor MyClass();
    constructor MyClass(a:Integer);
    m_a:Integer;
    m_b: real;
}

constructor MyClass()
{
    m_a=0;
    m_b=0.0;
}

constructor MyClass(a:Integer)
{
    m_a=a;
    m_b=0.0;
}

procedure main()
{
   var obj1:MyClass= { };
   var obj2:MyClass= {3*4};
}
```

6.5. Instrucciones de control de flujo

En LOOS tenemos implementadas instrucciones de control de flujo típicas como el If o los bucles While. Además de éstas, proponemos implementar las siguientes:

6.5.1. Instrucción do while [InstruccionDoWhile]

En este caso, a diferencia del bucle while do normal, siempre se ejecuta la instrucción al menos una vez, tal y como se ve en el ejemplo siguiente:

```
procedure main()
{
    var i:Integer;
    i=0;
    do
    {
        print "i=",i,"\\n";
        i=i+1;
    }
    while i>0;
}
```

6.5.1.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el do while. Se puede utilizar como modelo la instrucción while y modificarla convenientemente.

6.5.1.2. Análisis Semántica

El único análisis semántico corresponde a verificar que el resultado de la condición sea un booleano.

6.5.2. Instrucción Repeat until [InstruccionRepeatUntil]

Esta instrucción funciona de manera idéntica al do while propuesto anteriormente de modo que en este caso también se ejecuta la instrucción al menos una vez, tal y como vemos en el ejemplo siguiente:

```
procedure main()
{
    var i:Integer;
    i=0;
    Repeat
    {
          print "i=",i,"\\n";
        i=i+1;
    }
    until i>=10;
}
```

6.5.2.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el repeat until. Se puede utilizar como modelo la instrucción while y modificarla convenientemente.

6.5.2.2. Análisis Semántica

El único análisis semántico corresponde a verificar que el resultado de la condición sea un booleano.

6.5.3. Instrucción For to do [InstruccionForTo]

Esta variante del bucle for impone que la variable índice crezca positivamente desde el valor inicial a la izquierda del to hasta el valor final que aparece a la derecha del to, tal y como se muestra en el siguiente ejemplo:

```
procedure main()
{
    var i:Integer;
    for i=0 to 9 do
    {
        println "i=",i;
    }
    for var j=x+4 to y*2 do
    {
        println "j=",j;
    }
}
```

Es importante tener en cuenta en este caso que la variable índice j que va precedida por var ya se declara dentro del ámbito creado por el bucle for, no tenéis que controlar que la variable ya existiese de antes ya que se creará dentro del ámbito local. En caso contrario, la variable índice i ha de estar declarada. Cuidado que esta puede ser una variable global, local, local de otro ámbito o un campo de un objeto. Para simplificar no se admiten bucles donde el índice se especifique con algo más complicado que un identificador, por ejemplo:

```
for a[3]=0 to 9 do ...
for obj.c=10 to 20 do ...
```

6.5.3.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el for. Se puede utilizar como modelo la instrucción while y modificarla convenientemente.

6.5.3.2. Análisis Semántica

Las acciones semánticas por hacer para for son:

- Crear un ámbito local para el for (TS.NouAmbit(), TS.EliminarAmbit()) para el índice del for.
- Hay que comprobar que el resultado de las expresiones que calculan el valor inicial y final del índice sean de tipo integer.
- Si el for tiene var hay que declarar el índice como una nueva variable de tipo TInt.
- Si el for no tiene var hay que buscar en la tabla de símbolos el índice. Este ha de ser una variable de tipo TInt.

6.5.4. Instrucción For downto do [InstruccionForDownTo]

En este caso la variable índice decrece desde el valor inicial a la izquierda del to hasta el valor final que aparece a la derecha del to, tal y como se muestra en el siguiente ejemplo:

```
procedure main()
{
    var i:Integer;
    for i=9 downto 0 do
    {
        print "i=",i,"\\n";
    }
    for var j=9+x downto 3*2 do
    {
        print "j=",j,"\\n";
    }
}
```

Es importante tener en cuenta en este caso que la variable índice j que va precedida por var ya se declara dentro del ámbito creado por el bucle for, no tenéis que controlar que la variable ya existiese de antes ya que se creará dentro del ámbito local. En caso contrario, la variable índice i ha de estar declarada. Cuidado que esta puede ser una variable global, local, local de otro ámbito o un campo de un objeto. Para simplificar no se admiten bucles donde el índice se especifique con algo más complicado que un identificador, por ejemplo:

```
for a[3]=100 downto 9 do ...
for obj.c=10 downto 0 do ...
```

6.5.4.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el for. Se puede utilizar como modelo la instrucción while y modificarla convenientemente.

6.5.4.2. Análisis Semántica

Las acciones semánticas por hacer para for son:

- Crear un ámbito local para el for (TS.NouAmbit(), TS.EliminarAmbit()) para el índice del for.
- Hay que comprobar que el resultado de las expresiones que calculan el valor inicial y final del índice sean de tipo integer.
- Si el for tiene var hay que declarar el índice como una nueva variable de tipo TInt.
- Si el for no tiene var hay que buscar en la tabla de símbolos el índice. Este ha de ser una variable de tipo TInt.

6.5.5. Instrucción For else [InstruccionForElse]

La instrucción for else sigue la sintaxis del lenguaje C++ y añade la clausula else al estilo de Python. La sintaxis es

`for (`*`inicialización`*`; `*`condición`*`; `*`incremento`*`) `*`cuerpo`*` else `*`finalización`*

La inicialización, condición, incremento y else finalización son opcionales por lo tanto sería valido escribir

`for (;;) `*`cuerpo`*

La inicialización es una expresión o una declaración de variable, la condición es una expresión, el incremento es una expresión, el cuerpo es una instrucción y la finalización es una instrucción. Algunos ejemplos de for else son los siguientes:

```
procedure main()
{
    Var i:integer;
    for (i=0; i<10; i=i+1) PrintLn i;
    i=0;
    for ( ; i<10; ) {
       PrintLn i;
       I=i+1;
    }
    else PrintLn “Final”;
}
```

Otros ejemplos con ampliaciones de inicialización de variables, break y continue son los siguientes:

```
Var a:array[100] of integer;
Function buscar(x:integer):integer
{
    Var j=0;
    for (; j<100; j=j+1) {
        if a[j]==x then break;
    }
    else j=-1;
    return j; // retorna el índice donde encuentra x en el array a
              // o retorna si no lo encuentra -1
}

Procedure buscar2(x:integer)
{
    for (var j=0; j<100; j=j+1) {
        if a[j]==x then {
```

```
            PrintLn x, “ encontrado en “, j;
            Break;
    }
    else PrintLn x,” no encontrado”;
}
```

Como se puede ver el uso de break hace que no se ejecute el else del bucle for, así que la ejecución de for else empieza por la inicialización, luego en cada iteración del bucle ejecuta la condición, el cuerpo y el incremento. Si la condición no se cumple sale del bucle y ejecuta la finalización. En el caso que se ejecute un break sale del bucle sin ejecutar la finalización. La instrucción continue hace que se vaya a ejecutar la siguiente iteración del bucle. Esto supone que se ejecutará el incremento, la condición y el cuerpo si la condición se cumple.

6.5.5.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el for else. Se puede utilizar como modelo la instrucción while y modificarla convenientemente. Hay que considerar que incialización, condición, incremento y la clausula else finalización son opcionales. No hay que crear expresiones o declaraciones de variables nuevas ya que se pueden aprovechar las que ya existen en la gramática de LOOS.

6.5.5.2. Análisis Semántica

Las acciones semánticas por hacer para for else son:

- Las variables que se declaren en la inicialización pertenecen al ámbito del for else. Por lo tanto, son accesibles desde la condición, incremento, cuerpo y finalización. Así que hay que crear un ámbito local en la tabla de símbolos al empezar a compilar for else y eliminarlo al acabar la compilación de for else.
- En el caso que existan las instrucciones break o continue hay que crear un ámbito de bucle en la tabla de símbolos que incluya el cuerpo del bucle.
- Hay que verificar que el resultado de la condición sea un booleano.

6.5.5.3. Generación de código

La generación de código de for else se basa en saltos condicionales, incondicionales y la definición de etiquetas. Hay que definir una etiqueta para poder saltar al inicio del bucle (condición), una para salir del bucle ejecutando la finalización, otra que salga del bucle sin ejecutar la finalización para la instrucción break y una que salte al incremento para poder implementar la instrucción continue.

6.5.6. Instrucción Break [InstruccionBreak]

La instrucción break se utiliza para salir de cualquier bucle. Sólo es válida ponerla dentro de los bucles. En el caso que esté fuera de un bucle ha de dar error. Su sintaxis es `break;` . Algunos ejemplos de uso son los siguientes:

```
procedure main()
{
    Var i:integer;
    i=0;
    while i<10 do {
        if i==5 then break;
        PrintLn i;
        i=i+1;
    }
```

```
}
```

En este ejemplo imprimirá los números del 0 al 4 ya que al llegar a 5 sale del bucle. Como break funciona para todos los bucles hay que implementarla para los bucles while, do while, repeat until, for to, for downto, for else y otros. Sólo hay que considerar los bucles que existen en el lenguaje según las ampliaciones que se hagan en la práctica.

6.5.6.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el break. Como break solo es válida dentro de bucles se podría crear una nueva regla <instruccionConBreak> que sólo se utilizaría dentro de los bucles. Esta opción no se ha de implementar y dejamos la comprobación de si break está dentro o fuera de bucle al análisis semántico. Por lo tanto, solo hay que añadir break a <instruccio> y no preocuparse porque sintácticamente se acepte un break fuera de bucle.

6.5.6.2. Análisis Semántica

Como break solo se puede poner dentro de bucles en cada bucle se creará en la tabla de símbolos un ámbito de bucle. Esto supondrá poner en la tabla de símbolos un TSSeparadorBucle antes de compilar el cuerpo del bucle y eliminar este separador al acabar la compilación del cuerpo del bucle. Entonces cuando se compile break se verificará que en la tabla de símbolos haya un TSSeparadorBucle. Cuidado, que si hay antes un TSSeparadorFuncio quiere decir que el bucle esta fuera del cuerpo de la función que estamos compilando, y por lo tanto, el cuerpo de la función no forma parte del bucle y en este caso se ha de dar error de que break esta fuera de bucle. Ejemplo de código erroneo:

```
procedure main()
{
    Var i:integer;
    i=0;
    while i<10 do {
        procedure p() {
            break; // Error por break fuera de bucle.
         }
        PrintLn i;
        i=i+1;
    }
}
```

6.5.6.3. Generación de código

Para la generación de código break ha de buscar el separador de bucle en la tabla de símbolos y generar el código para saltar a la salida del bucle. Este código ha de considerar que se esta saliendo de ámbitos locales y que por lo tanto, hay que llamar a los destructores de las variables locales. Finalmente se utilizará un salto incondicional para continuar la ejecución después del bucle.

6.5.7. Instrucción Continue [InstruccionContinue]

La instrucción continue se utiliza para continuar la ejecución en la siguiente iteración del bucle. Se aplica a cualquier bucle. Sólo es válida ponerla dentro de los bucles. En el caso que esté fuera de un bucle ha de dar error. Su sintaxis es `continue;` . Algunos ejemplos de uso son los siguientes:

```
procedure main()
{
    Var i:integer;
    i=0;
    while i<10 do {
        i=i+1;
        if i==5 then continue;
        PrintLn i;
    }
}
```

En este ejemplo imprimirá los números del 1 al 10 saltando el 5 y ya que al llegar a 5 ejecuta la instrucción continue antes de imprimir el número. Esto hace que vaya a la siguiente iteración del bucle donde imprimirá el número 6. Como continue funciona para todos los bucles hay que implementarla para los bucles while, do while, repeat until, for to, for downto, for else y otros. Sólo hay que considerar los bucles que existen en el lenguaje según las ampliaciones que se hagan en la práctica.

6.5.7.1. Análisis Sintáctica

Se ha de modificar la regla <instruccio> añadiendo un caso más en la unión que será el continue. Como continue solo es válida dentro de bucles se podría crear una nueva regla <instruccionConContinue> que sólo se utilizaría dentro de los bucles. Esta opción no se ha de implementar y dejamos la comprobación de si continue está dentro o fuera de bucle al análisis semántico. Por lo tanto, solo hay que añadir continue a <instruccio> y no preocuparse porque sintácticamente se acepte un continue fuera de bucle.

6.5.7.2. Análisis Semántica

Como continue solo se puede poner dentro de bucles en cada bucle se creará en la tabla de símbolos un ámbito de bucle. Esto supondrá poner en la tabla de símbolos un TSSeparadorBucle antes de compilar el cuerpo del bucle y eliminar este separador al acabar la compilación del cuerpo del bucle. Entonces cuando se compile continue se verificará que en la tabla de símbolos haya un TSSeparadorBucle. Cuidado, que si hay antes un TSSeparadorFuncio quiere decir que el bucle esta fuera del cuerpo de la función que estamos compilando, y por lo tanto, el cuerpo de la función no forma parte del bucle y en este caso se ha de dar error de que continue esta fuera de bucle. Ejemplo de código erroneo:

```
procedure main()
{
    Var i:integer;
    i=0;
    while i<10 do {
        procedure p() {
            continue; // Error por continue fuera de bucle.
         }
        PrintLn i;
        i=i+1;
    }
}
```

6.5.7.3. Generación de código

Para la generación de código continue ha de buscar el separador de bucle en la tabla de símbolos y generar el código para saltar a la siguiente iteración del bucle. Este código ha de considerar que se está saliendo de ámbitos locales y que por lo tanto, hay que llamar a los destructores de las variables locales. Finalmente se utilizará un salto incondicional para continuar la ejecución en la siguiente iteración del bucle. La siguiente iteración del bucle empieza en el incremento o decremento que se hace al final del bucle y continua por la condición de salida del bucle.

6.5.8. Instrucción case of selector [InstruccionCase]

Esta instrucción realiza las mismas funcionalidades que un switch típico, tal y como vemos en el ejemplo siguiente:

```
procedure main()
{
    var i:integer;
    i=0;
    while i<5 do
    {
       case i of
       {
            0: print "caso 0\\r\\n";
            1: print "caso 1\\r\\n";
            2: print "caso 2\\r\\n";
            3: print "caso 3\\r\\n";
            else print "no caso\\r\\n";
       }
       i=i+1;
    }
}
```

Debéis tener en cuenta lo siguiente: después de cada caso (a la derecha de ":" pueden venir una, varias o ninguna instrucción. En el caso de varias instrucciones, éstas estarán agrupadas dentro de una llave, como vemos en el siguiente ejemplo:

```
procedure main()
{
    var i:integer;
    i=0;
    while i<5 do
    {
        case i of {
            0: {
                    print "caso 0\\r\\n";
                    print "primer caso\\r\\n";
               }
            1: print "caso 1\\r\\n";
            2: print "caso 2\\r\\n";
            3: print "caso 3\\r\\n";
            else print "no caso\\r\\n";
                 }
       i=i+1;
    }
}
```

En el caso de no tener ninguna instrucción asociada al caso, la sintaxis deberá ser la siguiente:

```
procedure main()
{
    var i:integer;
    i=0;
    while i<5 do
    {
        case i of
        {
           0: ;
           1: println "caso 1";
        }
        i=i+1;
    }
}
```

Finalmente, usamos el else como si fuera el otherwise o caso por defecto si no hemos entrado en ninguno de los anteriores. No es obligatorio ponerlo, pero se ha de considerar su efecto si aparece en el código.

6.5.8.1. Análisis Sintáctica

La instrucción case se ha de añadir como un caso más de la regla <instruccio>. El selector del case será una expresión y el cuerpo del case esta formado por una repetición de sentencias case y al final un else opcional.

6.5.8.2. Análisis Semántica

En la instrucción case hay que obtener el tipo de la expresión que hace selector y comprobar que se puede utilizar para la selección de casos. Esto supone comprobar que es un tipo de datos simple (integer, real, character y Bool) o un apuntador. La constante que pongamos antes del dos puntos en cada sentencia case ha de ser un literal. Se puede ver qué literales hay en <Factor>. Se tendrá que comprobar que los literales son del mismo tipo de datos que el selector.

6.5.9. Instrucción select [InstruccionSelect]

Esta instrucción realiza funcionalidades similares al switch aunque en este caso la condición para entrar en cada uno de los casos se encuentra definida por completo dentro de cada uno de ellos, tal y como vemos en el ejemplo siguiente:

```
procedure main()
{
    var i:integer;
    i=0;
    while i<5 do
    {
        Select
        {
            i<2 : print "caso i<2\\r\\n";
            i==2: print "caso i==2\\r\\n";
            i<4: print "caso i<4\\r\\n";
            else print "no caso\\r\\n";
        }
    i=i+1;
    }
```

```
}
```

De nuevo debéis tener en cuenta lo siguiente: después de cada caso (a la derecha de ":" pueden venir una, varias o ninguna instrucción. En el caso de varias instrucciones, éstas estarán agrupadas dentro de una llave, como vemos en el siguiente ejemplo:

```
procedure main()
{
    var i:integer;
    i=0;
    while i<5 do
    {
        Select
        {
          i<2 : {
                print "caso i<2\\r\\n";
                print "primer caso\\r\\n";
                }
          i==2: print "caso i==2\\r\\n";
          i<4: print "caso i<4\\r\\n";
          else print "no caso\\r\\n";
        }
        i=i+1;
    }
}
```

Finalmente, del mismo modo que en el **case selector of**, usamos el else como si fuera el otherwise o caso por defecto si no hemos entrado en ninguno de los anteriores. No es obligatorio ponerlo, pero se ha de considerar su efecto si aparece en el código.

6.5.9.1. Análisis Sintáctica

El cuerpo de la instrucción select está compuesta por una repetición de sentencias condicionales y al final opcionalmente la sentencia else.

6.5.9.2. Análisis Semántica

El análisis semántico del select consiste en verificar que el tipo de datos de las condiciones es un booleano.

6.5.10. Empleo de Bloques de Instrucciones [InstruccionBlock]

LOOS+ incorpora el uso de bloques de instrucciones. Dichos bloques aglutinan diversas instrucciones (siempre tiene que haber al menos una instrucción en el bloque), delimitadas por llaves. El uso de bloques viene acompañado por la instrucción `leave`, que permite salir de un bloque de instrucciones cuando dicha instrucción es invocada. El uso conjunto de bloques y la instrucción `leave` es especialmente útil cuando se necesita salir de funciones o instrucciones de control de flujo fuertemente anidadas (por ejemplo: tres bucles `for` anidados). Un ejemplo del uso correcto de la instrucción `Block` se muestra a continuación:

```
procedure main()
{
      var i:Integer;
      i=0;
      while i<5 do {
```

```
            Block B1 {
                  if i<3 then PrintLn i;
                  else Leave B1;
                  IMPRIMIR_TAULA_DE_SIMBOLS;
            }
            PrintLn "fora block";
            i=i+1;
      }
}
```

Es importante tener en cuenta que la instrucción `leave` debe ir seguida de un identificador que previamente haya sido definido como identificador de un bloque de instrucciones.

A nivel semántico, no se pueden declarar bloques anidados con el mismo nombre. Un bloque es visible en su interior excepto si definimos una función dentro de bloque. En este caso, no podremos salir del bloque con leave desde dentro de la función. Como dentro de la función ya no se ven los bloques exteriores a ella, se puede repetir el nombre de un bloque exterior sin que esto produzca un error de declaración duplicada.

En generación de código, cuando ejecutamos la instrucción leave, se salé de uno o más ámbitos locales, esto supone que además de saltar al final del bloque, tendrá que destruir los objetos declarados dentro de los ámbitos.

6.5.10.1. Análisis Sintáctica

Hay que implementar la sintaxis de la instrucción block y la instrucción leave. Estas son nuevos casos en la regla <Instruccio>. Sintácticamente no comprobaremos que leave está dentro de Block, esta comprobación se hará semánticamente. Dentro de la instrucción Block no conviene utilizar el símbolo no terminal <Bloc> ya que la estructura de la tabla de símbolos no coincidirá con la que se pide en el análisis semántico.

6.5.10.2. Análisis Semántica

En el análisis semántico de la instrucción Block hay que crear y eliminar su ámbito local y añadir una estrada en la tabla de símbolos del bloque. Esta entrada estará dentro del ámbito local de la instrucción Block. De esta forma solo se podrá acceder al block desde dentro de su ámbito local, ya que al salir del Block se elimina el ámbito del Block y con ello, la entrada de la tabla de símbolos del Block. La entrada de la tabla de símbolos de un block será ETSBlock que se declara como:

```
Type [public,Print=Contents] ETSBlock(
        Sortida // Etiqueta a la sortida del bloc
    ) from EntradaTaulaSimbols;
```

La creación de una entrada de bloque se hace con ETSBlock(nombre,etiqueta). La etiqueta se utiliza para generación de código y en análisis semántico será unbound.

No están implementadas las operaciones para comprobar si un Block esta duplicado o para ver si está declarado no están implementadas y se tienen que implementar para esta ampliación en el fichero que entreguéis (semantic.csl o gencod.csl). Podéis fijaros en cómo están implementadas las funciones que hacen búsquedas por la tabla de símbolos del fichero semantic.csm. Hay que tener en cuenta que el ámbito de un Block está limitado a la función donde se declara. Así que la detección del error de Block duplicado supondrá buscar por la tabla

de símbolos hasta encontrar un separador de función. Lo mismo si queremos saber si ya está declarado.

6.6. Operadores en expresiones

Aparte de las operaciones aritméticas típicas de suma, resta, multiplicación y división, añadiremos las operaciones siguientes:

6.6.1. Operadores módulo y división entera [OperadorDivMod]

El lenguaje LOOS+ deberá también incluir las operaciones módulo (`mod`) y división entera (`div`). Se recuerda que, en una división normal, la división entera correspondería al resultado de la división y el módulo al resto (por ejemplo: 5/3; la división entera daría como resultado 1 y el módulo daría 2).

```
var a:integer;
var b:integer;
a = 2;
b = 1;
var c:integer;
c = a mod b;
c = a div b;
```

6.6.1.1. Análisis Sintáctica

Div y Mod son operadores con la misma prioridad que el producto y la división.

6.6.1.2. Análisis Semántica

Los operandos de Div y Mod han de ser enteros y el resultado es un entero. El resultado será un RValue.

6.6.2. Operador valor absoluto [OperadorAbs]

Este operador calcula el valor absoluto de un número entero o real cualquiera, tal y como se muestra a continuación:

```
pocedure main()

{
     print "|50|=",|50|,"\\n";
     print "|-10*4|=",|-10*4|,"\\n";
}
```

6.6.2.1. Análisis Sintáctica

Como el valor absoluto encierra su operando entre "|", se pone en la regla <Factor> y su operando será una expresión. Sintácticamente es parecido a una expresión entre paréntesis.

6.6.2.2. Análisis Semántica

Sólo hay que comprobar que el operando es de tipo entero o real. El resultado será del mismo tipo que el parámetro y será un RValue.

6.6.3. Operadores binarios NOT, OR y AND [OperadorBoolBits]

Definimos los tres operadores lógicos básicos de números binarios, como se muestra en los ejemplos siguientes:

- NOT lógico:

```
procedure main()
{
     print "~100=",~100,"\\n";
}
```

- OR lógico:

```
procedure main()
{
     print "100 | 20=",100 | 20,"\\n";
}
```

- AND lógico:

```
procedure main()
{
     print "100 & 30=",100 & 30,"\\n";
}
```

La operación más prioritataria es not, la de prioridad intermedia es and y la menos prioritaria es el or.

6.6.3.1. Análisis Sintáctica

La operación de bits "|" tiene la misma prioridad que el "||", la operación de bits "&" tiene la misma prioridad que "&&" y la operación de bits "~" tiene la misma prioridad que "!".

6.6.3.2. Análisis Semántica

Las operaciones de bits tienen operandos de tipo entero y su resultado es un RValue entero.

6.6.4. Operador potencia [OperadorPow]

El lenguaje LOOS+ deberá también incluir la \textbf{operación potencia (**)}, implementada de manera similar al siguiente ejemplo:

```
var c:real;
c = 2**3;
```

En este caso tenéis que tener en cuenta la asociatividad de la operación potencia cuando os enfrentéis a su implementación.

6.6.4.1. Análisis Sintáctica

La operación "**" es más prioritaria que el producto, pero menos que el cambio de signo. Esto supone que se ha de crear una nueva regla para ella. Cuidado que tiene asociatividad por la derecha.

6.6.4.2. Análisis Semántica

La operación "**" tiene operandos de tipo real o entero y el resultado será un RValue de tipo real.

6.6.5. Operador máximo [OperadorMax]

Este operador calculará el máximo de dos o más valores numéricos, tal y como se muestra a continuación:

```
procedure main()
```

```
{
     print "max{20,10+f(x)}=",max{20,10+f(x)},"\\n";
     print "max{10.4,20.9}=",max{10.4,20.9},"\\n";
     print "max{'a','b','c'}=",max{'a','b','c'},"\\n";
}
```

Los operandos del operador max sólo pueden ser Integer, real y Character, y en una operación max han de ser del mismo tipo de datos.

6.6.5.1. Análisis Sintáctica

La operación max encierra sus operandos entre llaves. Esto supone que no hay que considerar problemas de prioridad con otros operadores. Entonces, para poder poner max en cualquier posición de una expresión lo mejor es darle la máxima prioridad. Un buen lugar donde poner es max es en la regla <factor>. Los operandos de max serán expresiones. Cuidado que max tiene como mínimo dos operandos.

6.6.5.2. Análisis Semántica

En la operación max hay que asegurar que todos los operandos son del mismo tipo de datos y el resultado será un RValue del tipo de los operandos. Una forma de hacerlo es obtener el tipo de datos de la primera expresión de max y comprobar que el tipo de datos del resto de las expresiones sea igual al de la primera.

6.6.6. Operador mínimo [OperadorMin]

Este operador calcula el mínimo de dos o más valores numéricos, tal y como se ve aquí:

```
procedure main()
{
     print "min{20,10}=",min{20,10},"\\n";
     print "min{10.5,4.6+r}=",min{10.5,4.6+r},"\\n";
     print "min{'a','b','c'}=",min{'a','b','c'},"\\n";
}
```

Los operandos del operador min sólo pueden ser Integer, real y Character, y en una operación min han de ser del mismo tipo de datos.

6.6.6.1. Análisis Sintáctica

La operación min encierra sus operandos entre llaves. Esto supone que no hay que considerar problemas de prioridad con otros operadores. Entonces, para poder poner min en cualquier posición de una expresión lo mejor es darle la máxima prioridad. Un buen lugar donde poner es min es en la regla <factor>. Los operandos de min serán expresiones. Cuidado que min tiene como mínimo dos operandos.

6.6.6.2. Análisis Semántica

En la operación min hay que asegurar que todos los operandos son del mismo tipo de datos y el resultado será un RValue del tipo de los operandos. Una forma de hacerlo es obtener el tipo de datos de la primera expresión de min y comprobar que el tipo de datos del resto de las expresiones sea igual al de la primera.

6.6.7. Operador Preincremento [OperadorPreincremento]

El operador preincremento funciona igual que en el lenguage C pero limitado a números enteros y reales y sólo se puede aplicar una vez a un operando. El operando del preincremento ha de ser un Left-Value (variable, elemento de un array, campo de una clase, etc.). Ejemplos:

```
++a;
c= ++b;
++obj.campo;
++t[i];
++p^.campo;
++f(x)^.campo;
c=a[++i];
```

En estos casos lo que hace es incrementar el valor de la variable (Left-Value) y su resultado es el valor incrementado que será un Right-Value, por lo que no se puede hacer ++ ++ a; ya que ++a no es un Left-Value. Un Left-Value es un elemento de memoria que contienen un valor. Por ejemplo, una variable, un campo de un objeto, un valor apuntado por un apuntador, etc. Un Right-Value es un valor que no tiene asociado una parte de memoria donde se guarde. Right-Value es un literal numérico, el resultado de una operación, el valor retornado por una función.

Usos incorrectos del preincremento:

```
++a=10;
++ ++a;
++f(x).campo;
```

6.6.8. Operador Postincremento [OperadorPostincremento]

El operador postincremento funciona igual que en el lenguage C pero limitado a números enteros y reales y sólo se puede aplicar una vez a un operando. El operando del postincremento ha de ser un Left-Value (variable, elemento de un array, campo de una clase, etc.). Ejemplos:

```
a++;
c= b++;
obj.campo++;
t[i]++;
p^.campo++;
f(x)^.campo++;
c=a[i++];
```

En estos casos lo que hace es obtener el valor de la variable y después incrementar la variable (Left-Value) y su resultado es el valor sin incrementar que será un Right-Value, por lo que no se puede hacer a ++ ++; ya que a++ no es un Left-Value. Un Left-Value es un elemento de memoria que contienen un valor. Por ejemplo, una variable, un campo de un objeto, un valor apuntado por un apuntador, etc. Un Right-Value es un valor que no tiene asociado una parte de memoria donde se guarde. Right-Value es un literal numérico, el resultado de una operación, el valor retornado por una función.

Usos incorrectos del preincremento:

```
a++=10;
a++ ++;
f(x).campo++;
```

6.6.9. Operador Predecremento [OperadorPredecremento]

El operador predecremento funciona igual que en el lenguage C pero limitado a números enteros y reales y sólo se puede aplicar una vez a un operando. El operando del predecremento ha de ser un Left-Value (variable, elemento de un array, campo de una clase, etc.). Ejemplos:

```
--a;
```

```
c= --b;
--obj.campo;
--t[i];
--p^.campo;
--f(x)^.campo;
```

En estos casos lo que hace es decrementar el valor de la variable (Left-Value) y su resultado es el valor decrementado que será un Right-Value, por lo que no se puede hacer -- -- a; ya que --a no es un Left-Value. Un Left-Value es un elemento de memoria que contienen un valor. Por ejemplo, una variable, un campo de un objeto, un valor apuntado por un apuntador, etc. Un Right-Value es un valor que no tiene asociado una parte de memoria donde se guarde. Right-Value es un literal numérico, el resultado de una operación, el valor retornado por una función.

Usos incorrectos del predecremento:

```
--a=10;
-- --a;
--f(x).campo;
```

6.6.10. Operador Postdecremento [OperadorPostdecremento]

El operador postdecremento funciona igual que en el lenguage C pero limitado a números enteros y reales y sólo se puede aplicar una vez a un operando. El operando del postdecremento ha de ser un Left-Value (variable, elemento de un array, campo de una clase, etc.). Ejemplos:

```
a--;
c= b--;
obj.campo--;
t[i]--;
p^.campo--;
f(x)^.campo--;
c=a[i--];
```

En estos casos lo que hace es obtener el valor de la variable y despues decrementar la variable (Left-Value) y su resultado es el valor sin decrementar que será un Right-Value, por lo que no se puede hacer a -- --; ya que a-- no es un Left-Value. Un Left-Value es un elemento de memoria que contienen un valor. Por ejemplo, una variable, un campo de un objeto, un valor apuntado por un apuntador, etc. Un Right-Value es un valor que no tiene asociado una parte de memoria donde se guarde. Right-Value es un literal numérico, el resultado de una operación, el valor retornado por una función.

Usos incorrectos del predecremento:

```
a--=10;
a-- --;
f(x).campo--;
```

7. Entregas

La realización de la práctica corresponde a implementar un subconjunto de todas las ampliaciones propuestas en este enunciado. Cada alumno tiene asignada un subconjunto de todas las ampliaciones. Podéis consultar el subconjunto que tenéis asignado en el campus virtual de la asignatura de Compiladors.

Se harán tres entregas principales y otras segundarias de la práctica en las que se irá completando la implementación de las ampliaciones del compilador. En la primera entrega principal sólo se ha de implementar la gramática de las ampliaciones (sintáctico). En la segunda entrega principal se implementas sobre la gramática las restricciones semánticas del lenguaje (Semántico). En la tercera entrega principal se añade la generación de código para las ampliaciones. Las entregas segundarias las definirá el profesor conforme avance en el temario de la asignadura.

Cada entrega está formada por un único fichero (sintactic.csl, semantic.csl o gencod.csl). En el fichero de entrega tenéis que identificaros escribiendo los siguientes datos de alumno:

- NIU
- nombre
- apellidos
- e-mail

Estos datos los tenéis que poner en la sección "Identificació dels alumnes del grup" del fichero CoSeL a entregar siguiendo la plantilla:

```
// ================================================================
// Identificació dels alumnes del grup ============================
// ================================================================

SetComponentsGrup([
   ("nom","cognoms","0000001","0000001@uab.cat")
])
```

La práctica se corregirá considerando el subconjunto de ampliaciones asignado al alumno que aparece en la identificación.

Las entregas se harán en el campus virtual de la asignatura de Compiladors. Estas entregas se abrirán como entregas individuales. Las fechas de cada entrega se indicarán en el campus virtual de la asignatura según con se vaya avanzando en el temario de la asignatura.

7.1. Prueba Práctica Antes de la Entrega

Como se ha comentado durante las diferentes sesiones de la asignatura, hemos implementado una web de autocorrección para la asignatura Compiladors. Esta web está accesible en http:\\compiladors.uab.cat. Los alumnos ya estáis prerregistrados usando como login vuestro mail del campus UAB (que suele ser nombre.apellido@e-campus.uab.cat). Una vez que entréis por primera vez en el sistema se os pedirá cambiar el password. El funcionamiento de la herramienta y la parte práctica es el siguiente: iréis haciendo ampliaciones en el fichero .csl correspondiente (sintactic.csl, semantic.csl o gencod.csl). Una vez que consideréis que habéis resuelto parte de la práctica, lo podéis comprobar subiendo el archivo .csl a la web donde se corregirá automáticamente. Esta corrección dará como resultado ficheros de texto que os

indicarán las ampliaciones que ya habéis completado, las que os quedan por completar y también os informará de la nota actual de la práctica.

7.2. Material disponible

El material disponible para realizar esta práctica es el siguiente:

- Enunciado de la práctica en Aula Moodle, sección Materiales Prácticas.
- Transparencias de teoría en Aula Moodle, sección Apuntes de Teoria.
- CrossVisions 2.4 en Aula Moodle, sección Software.
- Tutorial CoSeL en Aula Moodle, sección Software.
- Ayuda módulo de generación de compliadores, sección Software.
- Fichero Com.csm, sintactic.csl, Semantic.csm, GenCod.csm en Aula Moodle, sección Materiales Prácticas en dentro del fichero PracticaLOOS V?.zip.

Todo este material es susceptible de actualización. Así que tendríais que ir consultando el aula Moodle de compiladores por si aparecen nuevas versiones. Normalmente aparecen antes de empezar las sesiones de una nueva entrega.

7.3. Entrega Sintáctico

La sesión práctica de Análisis Sintáctico de la asignatura tendrá lugar durante las sesiones de clase normales de la asignatura. Durante estas contareis con el profesor para la resolución de dudas. También podéis hacer consultas en los horarios de tutoría i por e-mail. En esta entrega solo se considerarán los resultados correspondientes a las pruebas de análisis sintáctico. Las pruebas de análisis semántico y generación de código se han de ignorar.

El único entregable de esta sesión será el fichero sintactic.csl que deberá ser colgado en la entrega correspondiente del Aula Moodle. Los alumnos deberán tener en cuenta lo siguiente:

- Para que la práctica sea corregida el fichero NO ha de contener errores que impidan su carga correcta.
- **El fichero no será renombrado, tendrá que ser llamado: sintactic.csl**.
- En la plantilla se deberá sustituir los valores de NIU, nombre, apellidos y e-mail por los que correspondan al alumno.

IMPORTANTE: Tened en cuenta que, aunque hagáis varios intentos dentro de la herramienta de corrección automática **sólo se tendrá en cuenta para el cálculo de la nota final el fichero subido al Aula Moodle (campus virtual)**.

7.4. Entrega Semántico

La sesión práctica de Análisis Semántico de la asignatura tendrá lugar durante las sesiones de clase normales de la asignatura. Durante estas contareis con el profesor para la resolución de dudas. También podéis hacer consultas en los horarios de tutoría i por e-mail. En esta entrega se considerarán los resultados correspondientes a las pruebas de análisis sintáctico para recuperación y las de análisis semántico para la nota de la entrega. Las pruebas generación de código se han de ignorar.

El único entregable de esta sesión será un fichero semantic.csl que deberá ser colgado en la entrega correspondiente del Aula Moodle. Los alumnos deberán tener en cuenta lo siguiente:

- Para que la práctica sea corregida el fichero NO ha de contener errores que impidan su carga correcta.
- **El fichero no será renombrado, tendrá que ser llamado: semantic.csl**
- En la plantilla se deberá sustituir los valores de NIA, nombre, apellidos, e-mail y grupo de los alumnos por los valores de los componentes del grupo.

IMPORTANTE: Tened en cuenta que, aunque hagáis varios intentos dentro de la herramienta de corrección automática **sólo se tendrá en cuenta para el cálculo de la nota final el fichero subido al Aula Moodle**.

7.5. Entrega Generación de Código

La sesión práctica de Generación de Código de la asignatura tendrá lugar durante las sesiones de clase normales de la asignatura. Durante estas contareis con el profesor para la resolución de dudas. También podéis hacer consultas en los horarios de tutoría i por e-mail. En esta entrega se considerarán los resultados correspondientes a las pruebas de análisis sintáctico y semántico para recuperación y las de generación de código para la nota de la entrega. L

El único entregable de esta sesión será un fichero gencod.csl que deberá ser colgado en la entrega correspondiente del Aula Moodle. Los alumnos deberán tener en cuenta lo siguiente:

- Para que la práctica sea corregida el fichero NO ha de contener errores que impidan su carga correcta.
- **El fichero no será renombrado, tendrá que ser llamado: gencod.csl**
- En la plantilla se deberá sustituir los valores de NIA, nombre, apellidos, e-mail y grupo de los alumnos por los valores de los componentes del grupo.

IMPORTANTE: Tened en cuenta que, aunque hagáis varios intentos dentro de la herramienta de corrección automática **sólo se tendrá en cuenta para el cálculo de la nota final el fichero subido al Aula Moodle**.

7.6. Evaluación

Respecto a la evaluación, como sabéis tenemos unos test obligatorios y unos test adicionales. Para aprobar la práctica ha de pasar todos los test obligatorios. En el caso de fallar alguno de estos test obligatorios, en el fichero de texto asociado os indicaremos el test concreto para que podáis mejorar vuestro código. Esto no será así para el caso de los test adicionales, que usaremos para calcular la nota final de grupo. Estos test no son especialmente diferentes a los obligatorios, simplemente comprueban casos adicionales.

La nota final de la práctica será, por tanto:

- SUSPENSO si no se pasan todos los test obligatorios.
- 5 + 5*NumTestAdicionalesOK/NumTotalTestAdicionales en el caso de que se superen todos los test obligatorios.

La práctica es acumulativa, y por lo tanto, en la siguiente entrega se puede recuperar la todas entregas anteriores, pero con una penalización del 80% que se aplica de la siguiente forma:

nota practica= (max(nota recuperación, nota primera entrega)-nota recuperación)*0.8+nota primera entrega.

Además de las tres entregas de cada una de las partes de la práctica, habrá una entrega de recuperación. Para ver como se calcula la nota de la práctica, ver la presentación de la asignatura en el campus virtual.