uc3m

Procesadores del Lenguaje

Curso 2022-2023, campus de Colmenarejo

*Analizador del lenguaje **Basic Structs***

1 Introducción

La práctica consistirá en la implementación de un compilador capaz de analizar programas de un lenguaje denominado BS (Basic Structs), que contiene expresiones aritmético-lógicas, registros, arrays simples, funciones y estructuras de control básicas, comprobando la corrección de la entrada principalmente en términos de los tipos de los datos utilizados.

Tras conocer los fundamentos del tratamiento léxico, sintáctico y semántico y la familiarización con las herramientas de construcción de compiladores, la práctica permitirá conocer la implementación completa de un analizador en las fases, léxica, sintáctica y semántica, y emplear estos conocimientos para reconocer errores en el uso de tipos e informar al programador.

2 Tareas a realizar

La práctica se dividirá en las siguientes partes:

- Generación del analizador léxico (patrones léxicos) que reconozcan los elementos del lenguaje a este nivel.
- Generación de la gramática que permitirá reconocer el lenguaje proporcionado y los símbolos terminales y no terminales de la misma mediante el ***parser*** creado a partir de ésta.
- Añadir la capacidad para registrar y tratar variables declaradas por el programador, almacenando el tipo con el que se declaren.
    - **Se deberá crear una estructura tabla de símbolos capaz de registrar la información de las variables del programa, comenzando con las variables de tipos básicos y posteriormente las de tipo VECTOR. Solo es obligatorio reconocer vectores de 1 dimensión.**
    - **Adicionalmente, se precisará además una tabla de registros para almacenar los tipos complejos declarados en el programa, de tipo STRUCT**
- Comprobaciones semánticas de todas las expresiones del programa, incluyendo la inicialización de variables, las expresiones asignadas a variables, operadores aplicados y condiciones de los controles de flujo. Se harán comprobaciones de tipos y operaciones válidas para distinguir los diferentes tipos del lenguaje, incluyendo, cuando sea necesaria, la conversión de tipos compatibles.
- Opcionalmente, extender las tablas de variables y tipos para registrar las funciones declaradas y hacer comprobaciones semánticas en las llamadas a función.
- Opcionalmente, el compilador se recuperará de errores a diferentes niveles (léxicos, sintácticos y semánticos).

Hacer mínimo una de estas

NOTA: la máxima calificación podrá obtenerse solamente si se realiza al menos una de las partes opcionales indicadas.

2.1 Especificación del lenguaje de entrada

El lenguaje BSL (Basic Structs) es un pequeño lenguaje de programación imperativo fuertemente tipado. El lenguaje trabaja con varios tipos de datos básicos y permite definir tipos compuestos de tipo registro y arrays. Las estructuras de control de flujo básicas que contiene son el salto condicional y el bucle condicional

1. SENTENCIAS, VARIABLES Y TIPOS

Los tipos de datos que se incluyen son los números enteros (con base decimal o base octal, comenzando por "0", base hexadecimal, con "0X" o base binaria, "0B"), los números reales (incluyendo notación con punto decimal, "." y científica), los booleanos y los datos de tipo carácter. Sobre los números se ejecutarán operaciones de suma, resta, multiplicación y división ("+", "-", "*", "/"), tanto literales (ej. "5.13") como variables cuyo valor puede ser modificado a lo largo del programa, aplicando las prioridades habituales sobre los operadores aritméticos, con posibilidad de inversión de signo y además agrupación con paréntesis ("(",")"). Análogamente, las operaciones booleanas serán la conjunción ("AND", "&"), disyunción ("OR", "|") y negación ("NOT", "!"). También existirán operadores de comparación de valores numéricos de igual, menor, menor o igual, mayor, mayor o igual ("==", "<","<=", ">", ">="). Además, se permite la aparición de comentarios, dentro y fuera de las funciones, que deben ser tratados –e ignorados– por completo en el análisis léxico. Los comentarios comienzan con el carácter porcentaje '%' y se extienden hasta el final de la línea.

El lenguaje permite dos tipos de sentencias: evaluación de expresiones matemáticas y asignaciones a variables. Las asignaciones comienzan por un nombre de variable, seguido del signo igual "=", y terminan con una expresión matemática normal. En ningún caso se permitirá la asignación de valores en mitad de una expresión, es decir, serán incorrectas las sentencias del tipo:

```
A + 4 * B = 3 - 7 % Error: asignación (B=3) en una expresión
```

Las variables deberán haberse declarado con su tipo correspondiente antes de utilizarlas, además el lenguaje permite que declarar una lista de variables con el mismo tipo, separadas por comas. Todas las variables deberán haberse declarado con su tipo correspondiente antes de utilizarlas. El valor por defecto de las variables es cero si no se han inicializado. Debajo se incluye código de muestra con declaración de variables y asignación de expresiones aritméticas y lógicas:

```
% Declaraciones
REAL re1, re2
BOOLEANO bo % se asigna el resultado de una expresión

% Operaciones
```

```
re1=5 * +3 - 80/10 % esta expresión debe resultar 7
re2=10 / 5 * re1   % esta expresión debe resultar 4
bo=5<=3 %% comparación, debe resultar cierto
```

Como hemos visto, será posible declarar y usar variables de cada tipo. El tipo de una variable debe ser especificado en su declaración, y no se permiten re-declaraciones (por ejemplo, si la variable entera "re1" ha sido declarada una vez, no se podrá volver a declarar ninguna otra con el mismo nombre, tenga o no similar tipo).

```
REAL        miVariable % repetida
BOOLEANO    miVariable % error: mismo identificador, aunque
tenga otro
                       % tipo
```

IMPORTANTE: *los NOMBRES de las variables SON SENSIBLES A MAYÚSCULAS. Esto quiere decir que será legal declarar las siguientes cuatro variables en un mismo bloque, puesto que sus nombres NO SON EQUIVALENTES:*

```
real        miVariable
entero      MIVARIABLE
entero      MiVaRiAbLe
```

Cuando una variable es declarada sin asignársele un valor inicial, se le asignará el valor por defecto para su tipo de dato, de acuerdo a la siguiente tabla:

Tabla 1 Valores por defecto para variables sin inicializar

	Valor
ENTERO	0
REAL	0.0
BOOLEANO	FALSE

El lenguaje de entrada permite declarar variables de tipo VECTOR, con la sintaxis:

VECTOR <tipo> nombre [tam]
Donde <tipo> ha de ser ENTERO, REAL o BOOLEANO, y tam un número entero con el tamaño del vector. Por ejemplo:
VECTOR REAL posicion3D[3]

Puede accederse a la posición del vector con la operación habitual de indexado, entre 0 y long-1:

```
posicion3D[0]=x
posicion3D[1]=y
posicion3D[2]=z
```

Además, se puede acceder al tamaño de un vector con su propiedad "long"

```
dim=posicion3D.long
```

El acceso a un valor de índice del vector vendrá dado por la expresión para acceder al valor deseado:

```
ID [expr_ind]
```

Donde ID es la variable de tipo vector, y las expr_ind son expresiones enteras para obtener los índices correspondientes. Deben ser valores enteros que deberían estar dentro de los límites del vector, si bien esta comprobación no puede hacerse en tiempo de compilación. Los valores válidos para acceder a un vector en cada dimensión van desde 1 hasta el tamaño de cada dimensión.

Finalmente, pueden formarse compuestos de estilo registro, declarados como **REGISTRO**. El tipo compuesto disponible es tipo registro, con una serie de campos especificados con un tipo, nombre de variable(s) y finalizado con punto y coma:

REGISTRO nombre {tipo 1 campo1, campo2, ... ;
tipo 2 campoi, campoi+1,...;
...
tipo m campon;}

A continuación, mostramos un ejemplo de programa con esta construcción.

```
%------------------------------
%--- Programa de ejemplo ------
%------------------------------
%- Declaracion de variables ---
REAL re1
ENTERO en2 = 6 % con asignación de valor
BOOLEANO bo3 = 5<3 % se asigna el resultado de una expresión
CARACTER ca4 = 'h' % variable de tipo carácter, y su literal
REGISTRO CIRCULO {REAL cx, cy, radio; % definición de tipo
                                      % compuesto
                  CARÁCTER color;}    % definición de tipo
                                      % compuesto
CIRCULO circ1 % declaración variable struct (no permite
              % asignación al mismo tiempo que la declaracion


%%% Expresiones
5 + 6 % operacion
re1 = 3.7 % asignación simple
bo3 = bo3 < 7 AND 5.46+7*en2 | 4 % asignación compleja
circ1.cx = 0.0 % campo de variable struct
circ1.cy = 0.0 % campo de variable struct
circ1.radio = 10.0 % campo de variable struct
circ1.color = 'r' % campo de variable struct
```

Como hemos visto, será posible declarar y usar variables. El tipo de una variable debe ser especificado en su declaración, y no se permiten re-declaraciones (si la variable entera "en2" ha sido declarada una vez, no se podrá volver a declarar ninguna otra con el mismo nombre, tenga o no similar tipo).

2. REGLAS DE TIPOS

Podemos agrupar las reglas de tipado en dos categorías:

- Movimiento de datos de un punto (entrada) a otro (salida) donde se usa:

```
real a = 5.3 % asignación de un valor a la
             % variable "a"
SI cierto ENTONCES … % evaluación de una condición
             % de salto
```

- Varios datos que se combinan para producir un resultado (otro dato):

```
5 * 8.2 % multiplicación: hay que saber cómo se
        % combinan
        % estos valores y qué resultado producen
```

Ambos niveles están relacionados, a veces es necesario resolver una operación que combina dos datos, para saber el tipo de dato que se va a producir es válido para una aplicación posterior:

```
entero b = 5 + 8.2    % es necesario conocer el tipo de dato
                      % que genera la suma para saber si
                      % la asignación es legal
```

En todas las sentencias que mueven valores de un lugar a otro (p.ej. asignaciones), o que evalúan una expresión para usarla en un punto concreto (p.ej condiciones de salto), tanto el origen como el destino deben tener el mismo tipo de dato, o bien debe poderse realizar una conversión del tipo origen al tipo destino sin pérdida de datos.

En todas las sentencias que mueven valores de un lugar a otro (p.ej. asignaciones), o que evalúan una expresión para usarla en un punto concreto (p.ej argumentos de funciones o condiciones de salto), tanto el origen como el destino deben tener el mismo tipo de dato, o bien debe poderse realizar una conversión del tipo origen al tipo destino sin pérdida de datos.

Algunos tipos pueden transformarse automáticamente a otros, según la siguiente tabla de conversión:

Tabla 2 Conversión legal automática de tipos (sólo se incluyen conversiones directas)

Carácter	Entero	Se toma el valor número del carácter (0-255)
Entero	Real	Pasar de 32 bits que representan un entero en Ca2 a 32 bits que representan un real en IEEE754

De esta forma, las siguientes sentencias serán válidas:

```
real      r1 = 7.5          % OK: real -> real
entero    e1 = 'a'          % OK: caracter -> entero
real      r2 = e1           % OK: entero -> real
```

Pero no así los siguientes ejemplos:

```
entero    e1 = 7.5          % ERROR: real->entero
booleano  b1 = 7            % ERROR: entero -> booleano
booleano  b2 = r1           % ERROR: real->bool (r1 es de tipo
real)
```

Por otro lado, todas las operaciones que combinan/manejan dos o más valores deben asegurar que sus tipos son similares, o al menos que se puede realizar la conversión de uno de ellos al tipo del otro sin pérdida de datos. Respecto a las operaciones y expresiones, cada operador requiere uno varios tipos de dato concretos a la entrada, y devuelve así mismo un tipo específico de dato a la salida. En la siguiente tabla se especifican los tipos de datos compatibles con cada operador (sin tener en cuenta conversiones), y la salida correspondiente para cada tipo de entrada.

Tabla 3 Operadores y tipos de datos

	Tipo Origen	Tipo Resultado
MAS	Entero	Entero
	Real	Real
	Carácter	Carácter
MENOS	Entero	Entero
	Real	Real
	Carácter	Carácter
POR, ENTRE	Entero	Entero
	Real	Real
MAYOR, MENOR	Entero	Booleano
	Real	Booleano
	Carácter	Booleano
IGUAL	Entero	Booleano
	Real	Booleano
	Carácter	Booleano
	Booleano	Booleano
AND, OR	Booleano	Booleano

No obstante, cuando una de las entradas de un operador binario tiene un tipo A, y la otra entrada un tipo B, será necesario convertir una de ellas a la otra: el dato con tipo más restrictivo se transforma al tipo más general. Así, **Carácter** puede convertirse a **Entero** o **Real**, y **Entero** puede convertirse a **Real**. No se permite la conversión de **Booleano** a ningún otro tipo.

A continuación, se muestran varios ejemplos donde se combinan todas las reglas referentes al tipado. Las siguientes sentencias son válidas en BSL:

```
5.0*4.0                 % OK: real * real -> real
real r1 = 7.5+'c'*8     % OK: por el orden de operación, primero
se
                        % ejecuta 'c'*8 [carácter*entero-
>entero] y
                        % después la suma 7.5+<<resultado
anterior>>
                        % [real+entero->real]
```

Pero no así estas otras líneas:

```
a = 5.0 * ('d'<8)       % ERROR: aunque la primera operación es
válida
                        % 'd'<8 [carácter<entero, carácter se
convierte
```

```
                        % a entero y se comparan], su resultado
es un
                        % booleano, y 5.0*<<resultado>>
[real*booleano]
                        % es ilegal porque real no puede
convertirse
                        % directamente a booleano, ni al revés
entero e1 = 'c'+'c'     % OK: carácter -> entero
SI 5/7 ENTONCES         % ERROR: real -> booleano
```

3. FUNCIONES

Una función es un bloque de sentencias que, dados unos parámetros de entrada, realizan una serie de cálculos para obtener un único valor al que se denomina ***de retorno***.

Las funciones tienen un nombre que permite que sean referenciadas desde cualquier punto de un programa (de igual forma que se referencian las variables). Por tanto, para declarar una función es necesario especificar su nombre, el tipo y nombre de sus valores de entrada –argumentos–, el tipo de dato que tendrá su valor de retorno, y las sentencias que realizan los cálculos necesarios –cuerpo de la función–.

En nuestro lenguaje de programación, los argumentos no son obligatorios: una función puede recibir cero valores de entrada. Sin embargo, el cuerpo de una función debe contener, por lo menos, una sentencia.

Por ello, las sentencias que definen la declaración de una función son:

```
def_funcion::=FUNCION ID '(' lista_args ')' ':' KEYTIPO
                  '{' blq_sentencias '}'
```

Donde FUNCION y DEVOLVER son palabras reservadas, ID es el nombre que recibirá la función, KEYTIPO es el nombre del tipo de dato que devuelve la función, y el cuerpo de la función es un bloque de sentencias encerrado entre llaves.

Sobre la lista de argumentos, se define como una secuencia de argumentos separados por coma, pudiendo ser válida la lista vacía. Un argumento es una pareja compuesta por un tipo de dato y un nombre, similar a una declaración de variable sin asignación inicial de valor:

Una función es invocada cuando se escribe su nombre, seguida de una lista de expresiones entre paréntesis que coincide en número y tipo con sus argumentos:

```
uso_funcion       ::= ID '(' lista_expresiones ')'
lista_expresiones ::= lista_expresiones ',' expresion
                  |   expresion
```

El resultado será un valor con el tipo de dato de retorno de la función. Es importante notar que cada uno de los argumentos de la función puede ser una expresión completa, siendo legales sentencias como:

```
bla( 5.0, (8.0+b)*cubo(b)<7 )
```

En este caso se invocaría a la función `bla(real, booleano)`, o a la que coincida en nombre y número de argumentos, y tenga tipos de argumentos compatibles más cercanos a `(real, booleano)`.

4. CONTROL DE FLUJO

Finalmente, en cuanto al control de flujo, se definen dos estructuras de control: salto condicional y bucle condicional. El salto condicional es una sentencia que evalúa una expresión booleana –llamada condición–, y dependiendo de si su resultado es verdadero o falso, salta a un punto más avanzado del programa o sigue ejecutando las sentencias adyacentes.

Las sentencias BNF que describen la construcción de un salto condicional son las dos siguientes:

```
condicional ::= SI expresion ENTONCES blq_sentencias FINSI
              | SI expresion ENTONCES
                  blq_sentencias
                SINO
                  blq_sentencias
                FINSI
```

Como hemos indicado, SI, ENTONCES, SINO y FINSI son palabras reservadas del lenguaje. La primera variante descrita es un salto simple. Si la expresión de la condición evalúa a verdadero, se ejecutará el bloque de sentencias encerrado entre las palabras ENTONCES y FINSI y después se continuará con normalidad. En caso contrario, se saltará directamente a la primera instrucción a partir del fin del salto condicional.

La segunda opción es algo distinta. Si la condición evalúa a verdadero, se ejecutará el bloque de sentencias entre ENTONCES y SINO, y después se saltará a la sentencia siguiente al salto condicional. Si la condición evalúa a falso, se ejecutará el bloque entre SINO y FINSI, y luego se continuará con normalidad.

Un ejemplo de código con control de flujo de salto condicional se incluye a continuación:

```
%%% Sentencia condicional de control de flujo
SI re2==3 OR bo1 AND bo2 %% condición
ENTONCES %% cuerpo cuando la condición es verdadera
    re2 = re2-3
SINO %% cuerpo cuando la condición es falsa
    re2 = 10-re1*re1
FINSI %% fin de la sentencia
```

El bucle condicional es una sentencia que evalúa una expresión booleana –llamada condición–, y dependiendo de si su resultado es verdadero o falso, ejecuta el cuerpo del bucle o bien salta al final de éste. La estructura BNF correspondiente al bucle condicional es la siguiente:

```
bucle ::= MIENTRAS expresion blq_sentencias
          FINMIENTRAS
```

Siendo MIENTRAS, FINMIENTRAS palabras reservadas del lenguaje.

Para finalizar, las palabras reservadas del lenguaje tienen una condición especial y por tanto no pueden usarse como nombres de variables. Las palabras reservadas son:

Tabla 4 Palabras reservadas del lenguaje

PALABRAS RESERVADAS		
cierto	booleano	entonces
falso	vector	sino
entero	long	finsi
real	mientras	and
finmientras	si	or
not	registro	funcion
devolver		

3 Entrega Parcial

En una primera entrega deberá completarse la construcción del analizador léxico del lenguaje y la especificación de la gramática. Como se ha visto, el lenguaje de trabajo comprende símbolos (tokens) pertenecientes a diferentes tipos:

LEXER PARSER

SI
SI
SI SI

- Palabras reservadas Manual punto 4.3
- Identificadores Manual punto 4.4
- Números, con varios formatos (enteros con base decimal, binaria, octal y hexadecimal, reales con parte decimal y notación científica) Práctica anterior, devolverlo solo en un tipo que sea numero

SI
SI SI

- Operadores aritmético-lógicos AND,. OR,....
- Comentarios, los que comiencen por el símbolo "%" hasta fin de línea

En el ejemplo anterior vimos como implementar un reconocedor de expresiones aritméticas. Ahora, se solicita modificar la especificación del escáner para realizar tareas de reconocimiento de tokens del lenguaje objetivo del curso, separado del análisis sintáctico.

Se recomienda encapsular el escáner en una clase que se importe por el programa principal. Para ello, basta incluir los patrones y métodos definidos para lex en una clase independiente:

```
import ply.lex as lex
import sys

class MyLexer():
  # CONSTRUCTOR
  def __init__(self):
    print('Lexer constructor called.')
    self.lexer = lex.lex(module=self)

  # DESTRUCTOR
  def __del__(self):
    print('Lexer destructor called.')

  #variables dentro de la clase
  tokens = (
    NUM', 'VAR',
  )

  # funciones dentro de la clase
  # todos los metodos deben tener como primer argumento "self"
  def t_error(self,t):
    print('Illegal character '%s' % t.value[0])
    t.lexer.skip(1)
....
```

Y a continuación se puede importar la clase en el programa principal:

```
import sys
sys.path.insert(0, "../..")

from myLexer import MyLexer
# create objects MY LEXER and MY PARSER
myLex = MyLexer()
lexer=myLex.lexer

...
```

El lenguaje con el que vamos a trabajar hasta el resto del curso comprende símbolos terminales (tokens) pertenecientes a diferentes tipos:

- Palabras reservadas
- Identificadores (deben comenzar con una letra, y después pueden tener letras, números o el carácter "_")
- Números, con varios formatos (enteros con base decimal, octal y hexadecimal, reales con parte decimal y notación científica)
- Operadores aritmético-lógicos
- Separadores, paréntesis y operador de índice de array: ";()[]"

(además, se mantienen los comentarios de una sola línea y multi-línea)

Como se ha visto, las sentencias del programa estarán separadas por saltos de línea, reservándose el finalizador ";" únicamente para los campos de la construcción STRUCT.
Respecto a las palabras reservadas del lenguaje, como se ha indicado en la tabla anterior, no pueden utilizarse en nombres de identificadores (variables, tipos o funciones)

Nota2: Las palabras reservadas pueden escribirse en mayúscula, minúscula o cualquier combinación.

Nota3: Se recomienda revisar las recomendaciones acerca del reconocimiento de palabras reservadas indicadas en la documentación de PLY.

ENTREGA

Cada grupo debe entregar el contenido de su práctica en un único archivo comprimido –preferiblemente en formato ZIP–. El nombre del comprimido debe ser "`pl_ grupo_XX`", (pe-ele_) donde XX son los apellidos de los integrantes del grupo.
El código fuente del proyecto incluirá al menos:

- Archivo .lex/.flex con la especificación de un scanner que sirva a los propósitos de la práctica.
- Clases auxiliares necesarias para ejecutar el analizador léxico y visualizar todos los tokens en la entrada.
- Diseño de la gramática con la estructura sintáctica del lenguaje (sin necesidad de hacerse funcionar aún en cup)

4 Entrega final

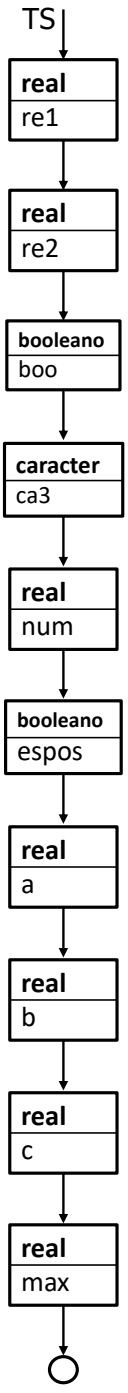
En la entrega final, la práctica deberá contener las tareas indicadas (scanner, parser, tabla de símbolos que almacene variables y tipos y análisis semántico). La gramática debe ser diseñada de tal manera que el parser no tenga conflictos de ningún tipo.

Se muestran a continuación varios ejemplos para ilustrar las representaciones de tipos que puede manejar el lenguaje y las comprobaciones que deben llevarse a cabo:

Ejemplo 1

```
<!-- Operaciones aritméticas y comparación
-----------------------------------------------
-----------------------------------------------
-->
REAL re1, re2
BOOLEANO bo
CARACTER ca = 'h' % variable de tipo carácter, y su literal

re1 =5 * 3 - 80/10 % esta expresion debe resultar 7
re2 =10 / 5 * re1     % esta expresion debe resultar 4
bo =5<3 % comparación, debe resultar cierto


% si un número es positivo
REAL num
BOOLEANO espos
SI num >= 0 ENTONCES
   Espositivo = CIERTO
SINO
   Espositivo = FALSO
FINSI

%cálculo del máximo
REAL a,b,c,max
a =1
b =2
c =3
SI a>=b AND a>=c ENTONCES % a es el máximo
   max =a
SINO % el máximo es b o c
   SI b>=a AND b>=c ENTONCES % b es el máximo
         max =b
   SINO % c es el máximo, por descarte
         max =c
   FINSI
FINSI
```

En este ejemplo todas las variables son de tipos básicos predefinidos (real, entero, booleano, carácter). A continuación, se muestra la lista completa de símbolos (TS) almacenada con una estructura de tipo lista.



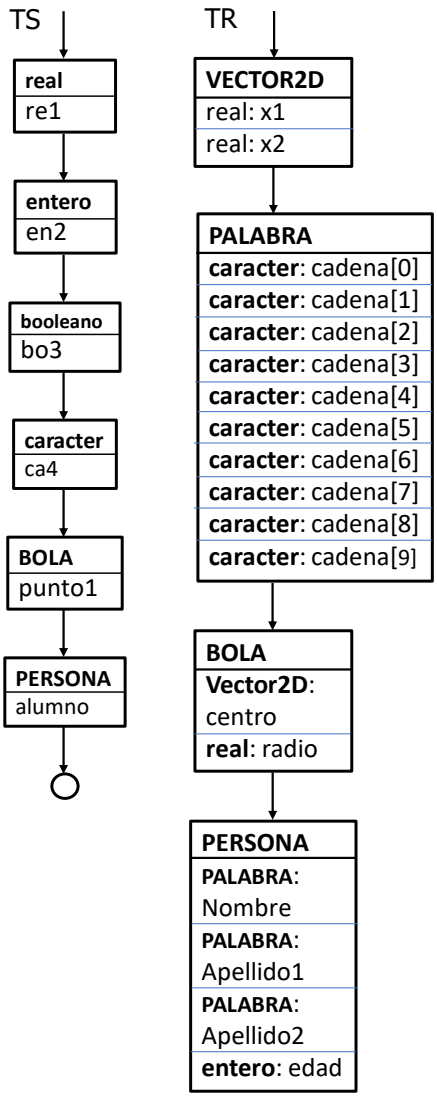
Ejemplo 2

Ejemplo con datos de tipo STRUCT, incluyendo VECTOR:

```
<!-- Declaracion de variables y tipos STRUCT
-----------------------------------------------
-----------------------------------------------
-->

REAL re1
ENTERO en2 = 6 % con asignación de valor
BOOLEANO bo3 = 5<3 % se asigna el resultado de una expresión
CARACTER ca4 = 'h' % variable de tipo carácter, y su literal
STRUCT VECTOR2D {REAL x1, x2;}
STRUCT BOLA {VECTOR2D centro; REAL radio;} %definición tipo
compuesto
BOLA punto1 % declaración variable struct (no permite
asignación al
%               mismo tiempo que la declaracion
STRUCT PALABRA {VECTOR CARACTER cadena[10];}
STRUCT PERSONA {PALABRA nombre, apellido1, apellido2;
                ENTERO edad;}
PERSONA alumno


%%% Expresiones
5 + 6 % operacion
re1 = 3.7 % asignación simple
bo3 = en2 < 7 AND 5.46+7*en2 > 4 % asignación compleja
punto1.centro.x1 = 0.0 % campo de variable struct anidada
punto1.centro.x2 = 0.0 % campo de variable struct anidada
punto1.radio = 10.0 % campo de variable struct

alumno.nombre.cadena[0]='A'
alumno.nombre.cadena[1]='l'
alumno.nombre.cadena[2]='b'
alumno.nombre.cadena[3]='e'
alumno.nombre.cadena[4]='r'
alumno.nombre.cadena[5]='t'
alumno.nombre.cadena[6]='o'
alumno.nombre.cadena[7]=0

alumno.apellido1.cadena[0]='G'
alumno.apellido1.cadena[1]='a'
alumno.apellido1.cadena[2]='r'
alumno.apellido1.cadena[3]='c'
alumno.apellido1.cadena[4]='í'
alumno.apellido1.cadena[5]='a'
alumno.apellido1.cadena[6]=0

alumno.apellido2.cadena[0]='S'
alumno.apellido2.cadena[1]='a'
alumno.apellido2.cadena[2]='n'
alumno.apellido2.cadena[3]='z'
alumno.apellido2.cadena[4]=0

alumno.edad=19


%%% Sentencia de control de flujo bucle condicional
```

```
BOOLEANO seguir= cierto
MIENTRAS seguir AND punto1.radio>0 % bucle condicional
SI punto1.centro.x1 <0 AND punto1.centro.x2 <0 % cuerpo si
condición verdadera
    seguir = false
SINO % cuerpo cuando la condición es falsa
    Seguir = cierto
FINSI % fin de condicional
punto1.radio = punto1.radio/2.0
punto1.centro.x1 = punto1.centro.x1 - 1.0
punto1.centro.x2 = punto1.centro.x2 - 1.0

FINMIENTRAS % fin de bucle MIENTRAS
```

En este segundo ejemplo tenemos variables declaradas de tipos construidos en el programa (punto1 de tipo BOLA, alumno de tipo PERSONA). Por tanto, además de la tabla de símbolos con todas las variables, necesitamos una tabla de registros (TR) con los tipos que forman las estructuras declaradas. Obsérvese que el tipo PALABRA es una construcción intermedia que permite almacenar una cadena de hasta 10 caracteres.

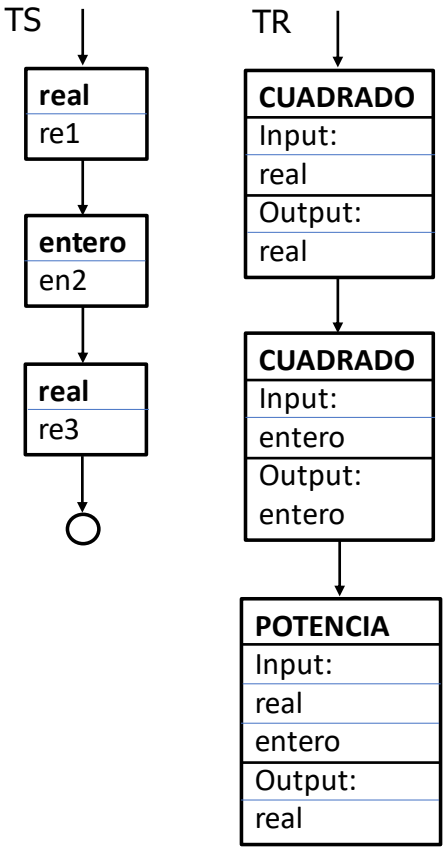


Ejemplo 3:

Ejemplo con declaración y uso de funciones:

```
<!—Definición de funciones
-----------------------------------------------
-----------------------------------------------
-->
FUNCION cuadrado( REAL a ): REAL
{
    DEVOLVER a * a
}
%%% Sobrecarga de funciones
FUNCION cuadrado( ENTERO e ): ENTERO
{
    DEVOLVER e * e
}

%%% Funcion de dos argumentos
FUNCION potencia( REAl a, ENTERO e ): REAL
{
    REAL r =1
    ENTERO i =1
    MIENTRAS i<e
      r = r*a
      i = i+1
    FINMIENTRAS
    DEVOLVER r
}


ENTERO en2 = 6 % con asignación de valor

%%% Expresiones
5 + 6 % operacion
re1 = 3.7 % asignación simple
re1 = 3.7 + cuadrado(re1) % usando llamada a función
re2 = potencia(re1,en2)

%%% Sentencia condicional de control de flujo
SI en2==3 OR bo3 AND re1 % condición
ENTONCES % cuerpo cuando la condición es verdadera
    en2 = en2+3
SINO % cuerpo cuando la condición es falsa, llamada recursiva
    re1 = cuadrado( cuadrado (en2) )
FINSI % fin de la sentencia
```

En este tercer ejemplo tenemos variables de tipos básicos, y además tres funciones declaradas (CUADRADO, CUADRADO, POTENCIA), las dos primeras con sobrecarga de nombres. Por tanto, además de la tabla de símbolos con todas las variables, necesitamos una tabla de registros (TR) que refleje las funciones, con los tipos que forman la entrada (Input) y salida (Output)


TS
real
re1
entero
en2
real
re3
TR
CUADRADO
Input:
real
Output:
real
CUADRADO
Input:
entero
Output:
entero
POTENCIA
Input:
real
entero
Output:
real

4.1 Especificación de la salida

El programa generará información de los errores de tipos del programa, indicando claramente los tipos de error, tales como variables no declaradas, asignación de tipo incompatibles, usar un campo de un ***Struct*** en una expresión invalida, errores de tipos en expresiones o condiciones, llamada a función con tipos incorrectos, etc.

En caso de no tener errores, la salida del programa consistirá en un archivo que liste las variables declaradas y sus tipos. En el caso de no existir estructuras de control de flujo ni llamadas a funciones, el programa además mostrará los valores de las variables tras la ejecución del programa.

4.2 Entrega

Cada grupo debe entregar todo el contenido de su práctica en un único archivo comprimido –preferiblemente en formato ZIP–. El nombre del comprimido debe ser "`pl_ grupo_XX`", (pe-ele_) donde XX son los apellidos de los integrantes del grupo. El código fuente del proyecto incluirá al menos:

- Archivo .lex/.flex con la especificación de un scanner que sirva a los propósitos de la práctica.
- Archivo .cup con la especificación del parser que, trabajando conjuntamente con el scanner anterior, satisfaga los requisitos de la práctica.
- Clases auxiliares necesarias para implementar el compilador (tabla de símbolos, comprobaciones semánticas, etc.)

- Un conjunto de archivos con programas expresiones aritméticas correctamente formadas.
    - Las expresiones correctas deben demostrar el correcto funcionamiento del programa y su traducción a código intermedio.
- En caso de implementar las partes opcionales, se deberá entregar otro conjunto de archivos con pruebas específicas para probar la funcionalidad adicional
    - Las expresiones incorrectas deben demostrar que el programa falla con casos no cubiertos en el archivo que se proporciona con el código base.

El archivo comprimido también incluirá una memoria en formato PDF con, al menos, los siguientes apartados:

- **Portada:** asignatura, práctica, composición y nombre del grupo, año lectivo.
- **Tabla de contenidos:** secciones y números de página correspondientes.

- **Introducción**
- **Gramática:** versión final de la gramática implementada, en notación BNF.
- **Breve descripción de la solución:** explicar la solución implementada y los controles semánticos realizados.
- **Archivos de prueba:** explicación de los aspectos probados en los archivos de prueba entregados.
- **Conclusiones:** aspectos adicionales a tener en cuenta por el corrector, y hechos a destacar sobre desarrollo de la práctica (incluyendo, pero no limitándose, a opiniones personales sobre el material de partida, conocimientos y cantidad de trabajo necesaria para su consecución).

4.3 Consideración final

Es muy importante respetar de forma estricta el formato de entrega. De no hacerlo, se corre el riesgo de perder puntos de calificación. Cada grupo responderá de forma totalmente responsable del contenido de su práctica. Esto implica que los autores deben conocer en profundidad todo el material creado por ellos.

Ante dudas de plagio o ayuda externa, el corrector convocará al grupo para responder a cuestiones sobre cualquier aspecto de la memoria o código entregados.