



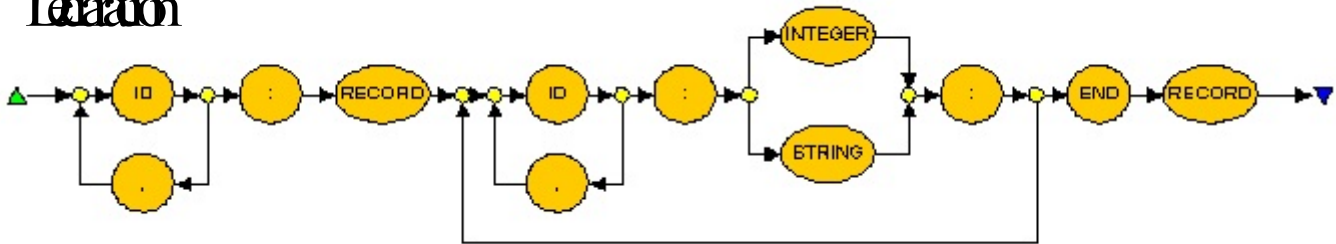
Apellidos, Nombre: _____

Calificación: _____

TEORÍA

1.- Partiendo del siguiente diagrama de sintaxis

Datación



se pide obtener la función en C equivalente desde el punto de vista del análisis sintáctico.

2.- Sea la gramática:

$$S \rightarrow h Q b p$$

$$Q \rightarrow b p$$

$$| b p Q$$

responder a las preguntas justificando las respuestas:

- ¿Es ambigua?
- ¿Es LALR(1)?

3.- Hacer un programa Lex cualquiera que reconozca números naturales y que diga si el número es múltiplo de 3 o no. No se permite usar la operación de división ni la de módulo en C. Es recomendable utilizar condiciones start (estados léxicos).



Apellidos, Nombre: _____
Calificación: _____

PRÁCTICA

Se pretende construir un intérprete para el lenguaje Bitipore. Este lenguaje permite la **declaración de variables** y su posterior **utilización en expresiones y asignaciones**.

TIPOS.

Este lenguaje permite el uso de los siguientes tipos:

- Dos tipos primitivos:
 - **INTEGER** que se corresponde con un **int** de C.
 - **STRING** que se corresponde con un **char*** de C.
- El tipo **RECORD** que permite crear registros cuyos campos sólo pueden ser de tipos primitivos, o sea, un registro no puede contener registros.

DECLARACIONES.

Las declaraciones pueden aparecer en cualquier lugar del código e incluso mezclado con éste, pero una variable nunca puede usarse antes de que sea declarada, ya que ello supone un error.

Las declaraciones son de la forma:

v1, v2, ..., vN : TIPO;

donde TIPO es INTEGER, STRING, o bien un registro de la forma:

```
RECORD
    v11, v12, ..., v1k : TIPO_PRIMITIVO;
    v21, v22, ..., v2k : TIPO_PRIMITIVO;
    ...
    v31, v32, ..., v3k : TIPO_PRIMITIVO;
END RECORD
```

Ejemplos de declaraciones son:

```
a : INTEGER;
b, c : STRING;
d : RECORD
    nombre, apellidos : STRING;
    nif : INTEGER;
END RECORD;
```

Como puede verse, el usuario **no** puede crear tipos nuevos (como por ejemplo se hace en C con la cláusula **typedef**), sino sólo declarar un RECORD en el mismo punto en el que se declarn las variables de ese tipo registro.

SENTENCIAS.

Hay dos tipos de sentencias, la sentencia de asignación y la sentencia de visualización:

- La asignación es de la forma


```
ASSIGN IValor = expresión;
```

 pudiendo ser IValor tanto una variable de tipo primitivo como un campo de una variable de tipo registro. Asignaciones válidas son:

```

ASSIGN a = 23;
ASSIGN b = "Hola";
ASSIGN b = b + b;
ASSIGN d.nombre = "Pepe";
ASSIGN d.apellidos = "Garcia Dato";
ASSIGN d.nif = 2765;

```

En cuanto a las expresiones, se permite el producto de enteros y la suma de enteros y/o cadenas de caracteres; la suma de dos cadenas equivale a la concatenación, y si se suman una cadena y un entero (o viceversa), el entero se convierte a texto y se concatena con la cadena. Cualquier otra combinación de tipos de datos en estas operaciones desencadena un error.

- La visualización es de la forma:

```
PRINT expresión;
```

y visualiza por pantalla cualquier expresión válida de tipo entera o cadena de caracteres.

Ejemplos de sentencias PRINT son:

```

PRINT a;
PRINT a*2+6;
PRINT b;
PRINT b + a + "Adios";
PRINT d.nombre + d.apellidos;
PRINT d.nif + a;

```

SINTAXIS.

La sintaxis completa en YACC para este problema es la siguiente:

```

prog : /* Epsilon */
      | prog decl ';'
      | prog sent ';'
      | prog error ';'
      ;
decl : ID ':' tipo
      | ID ',' decl
      ;
tipo : tipoBase
      | RECORD listaDeclBasica END RECORD
      ;
tipoBase : INTEGER
          | STRING
          ;
listaDeclBasica : declBasica ';'
                 | listaDeclBasica declBasica ';'
                 | error ';'
                 | listaDeclBasica error ';'
                 ;
declBasica : ID ':' tipoBase
            | ID ',' declBasica
            ;
sent : PRINT expr
      | ASSIGN lValor '=' expr
      ;
lValor : ID
        | ID '.' ID
        ;
expr : lValor
      | NUM
      | CADENA
      | expr '+' expr
      | expr '*' expr

```

Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes

OCW- Universidad de Málaga <http://ocw.uma.es>

Bajo licencia [Creative Commons Attribution-Non-Comercial-ShareAlike](https://creativecommons.org/licenses/by-nc-sa/4.0/)

```
| '(' expr ')'  
;
```

EJEMPLO

A continuación se muestra un ejemplo de entrada y la salida que debe producirse:

Entrada	Salida
PRINT 23;	23
PRINT 23+5;	28
PRINT "ALDA";	ALDA
PRINT "ALFA"+"BETA";	ALFABETA
PRINT ("ALFA"+23)+"NUNO";	ALFA23NUNO
a : INTEGER;	
b, c : STRING;	
d : RECORD	
nombre, apellidos : STRING;	
nif : INTEGER;	
END RECORD;	
ASSIGN a = 23;	
PRINT a;	23
PRINT a*2+6;	52
ASSIGN b = "Hola";	
ASSIGN b = b + b;	
PRINT b;	HolaHola
PRINT b + a + "Adios";	HolaHola23Adios
ASSIGN d.nombre = "Pepe";	
ASSIGN d.apellidos = "Garcia Dato";	
ASSIGN d.nif = 2765;	
PRINT d.nombre + d.apellidos;	PepeGarcia Dato
PRINT d.nif + a;	2788
PRINT a.nombre;	Esperaba un registro. Linea 25
PRINT "A"*2;	Producto imposible. Linea 26
PRINT "A"*2*"A";	Producto imposible. Linea 27
	Producto imposible. Linea 27
PRINT d.trasgo;	El campo trasgo no esta en d. Linea 28
PRINT d;	Esperaba una variable de tipo primitivo. Linea 29
ASSIGN x=7;	x no es una variable. Linea 30
	Asignacion imposible. Linea 30
PRINT y;	y no es una variable. Linea 31

TABLA DE SÍMBOLOS

Para realizar todo esto, se suministra una tabla de símbolos que sirve para almacenar el tipo y el valor de cada variable. En el caso de las variables de tipo registro se guarda además un puntero a la estructura de definición del registro, así como los valores de los campos en dos arrays, uno para los datos de tipo entero, y otro para los datos de tipo cadena de caracteres. El fichero que define la tabla de símbolos se muestra a continuación:

Fichero TabSim05.c

```
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
typedef struct _nodoNombre{  
    char nombre[20];  
    struct _nodoNombre * sig;  
} nodoNombre;
```

Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes

OCW- Universidad de Málaga <http://ocw.uma.es>
Bajo licencia [Creative Commons Attribution-Non-Comercial-ShareAlike](https://creativecommons.org/licenses/by-nc-sa/4.0/)

```

// nodoRecord sirve para definir la estructura lógica de un registro. Desde
// este punto de vista, un registro es una secuencia de nombres de campos de
// tipo entero y otra secuencia de nombres de campos de tipo cadena.
typedef struct _nodoRecord {
    nodoNombre * enteros;
    nodoNombre * cadenas;
} nodoRecord;

// nodoDato sirve para guardar el tipo y el valor de cada variable, tanto si
// es de tipo entero, cadena o registro. En caso de ser de tipo entero o
// cadena
// el funcionamiento es muy parecido al visto en clase.
// Si se trata de un registro, entonces hay que guardar el valor de cada campo
// de tipo entero, y el valor de cada campo de tipo cadena. Para ello se usan
// arrays (valoresInteger y valoresString) -recordemos que un array es un
// puntero- uno que guarda los valores de los campos de tipo entero y otro
// para
// guardar los valores de los campos de tipo cadena. Así, por ejemplo, la
// posición i-ésima del array de enteros almacena el campo entero i-ésimo del
// registro, cuyo nombre estará en la posición i-ésima de la secuencia de
// nombres
// de campos enteros del nodoRecord (registro) que define la estructura del
// Dato.
typedef struct _nodoDato {
    char nombre[20];
    char tipo;
    union{
        int valorInteger;
        char * valorString;
        struct{
            nodoRecord * registro;
            int * valoresInteger;
            char * * valoresString;
        } valorRecord;
    } valores;
    struct _nodoDato * sig;
} nodoDato;

nodoDato * tablaDatos;

void insertarDato(nodoDato * x){
    x->sig = tablaDatos;
    tablaDatos = x;
}

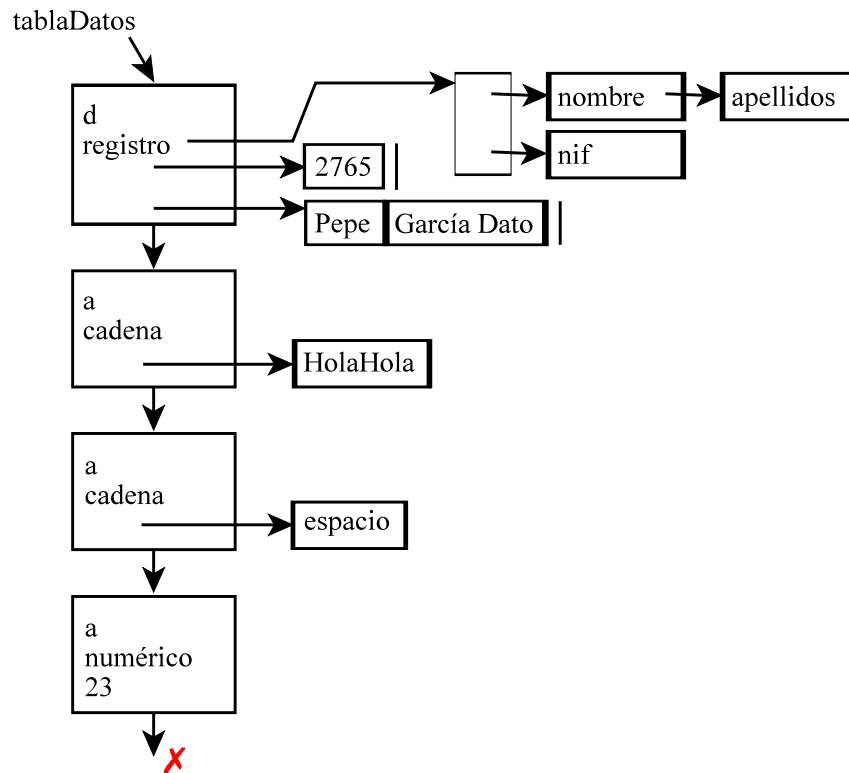
nodoDato * buscarDato(char *nombre){
    nodoDato * t = tablaDatos;
    while( (t != NULL) && (strcmp(nombre, t->nombre)) )
        t = t->sig;
    return (t);
}

// Estas funciones sirven para buscar el nombre de un campo en un registro y
// devolver su posición en la secuencia de nombres. Esta posición servirá,
// posteriormente, para acceder al valor de dicho campo que estará en un array
// de valores dentro del nodoDato.
int buscarIntegerEnRecord(char * nombre, nodoRecord* registro){
    return buscarNombreEnListaDeNombres(nombre, registro->enteros);
}
int buscarStringEnRecord(char * nombre, nodoRecord* registro){
    return buscarNombreEnListaDeNombres(nombre, registro->cadenas);
}
int buscarNombreEnListaDeNombres(char * nombre, nodoNombre * lista){
    int pos = 0;
    while( (lista != NULL) && (strcmp(nombre, lista->nombre)) ){
        pos++;
        lista = lista->sig;
    }
    return (lista==NULL)?-1:pos;
}

```

}

Para ilustrar esto, a continuación se muestra como queda la tabla de símbolos tras interpretar el programa de ejemplo anterior:



En base a todo lo anterior, se pide:

- Rellenar el esqueleto Lex/Yacc propuesto para conseguir las funcionalidades pedidas. No es necesario liberar memoria con free.

Fichero ExDic05l.lex

%%

Fichero ExDic05y.yac

```
%{
#include "tabsim05.c"
// Funciones y variables auxiliares
nodoRecord * registroActual;
void reservaMemoria(nodoDato * nuevo){
```

```

nuevo->valores.valorRecord.valoresInteger=
    (int *)malloc(sizeof(int)*longitud(nuevo->valores.valorRecord.registro->enteros));
nuevo->valores.valorRecord.valoresString=
    (char**)malloc(sizeof(char*)*longitud(nuevo->valores.valorRecord.registro->cadenas));
}
int longitud(nodoNombre * ptr){
    int ret;
    for(ret=0; ptr!=NULL; ptr=ptr->sig, ret++);
    return ret;
}
%}
%union{
    int valorInteger;
    char * valorString;
    char nombre[20];
    char tipo;
    struct {
        char tipo;
        union{
            int valorInteger;
            char * valorString;
        } valor;
    } tipoYValor;
    struct {
        char tipo;
        union{
            int * ptrInteger;
            char * * ptrString;
        } ptr;
    } tipoYPuntero;
}
%token < > ID
%token RECORD END INTEGER STRING PRINT ASSIGN
%token < > NUM
%token < > CADENA
%type < > tipo tipoBase decl declBasica
%type < > expr
%type < > IValor
%left '+'
%left '*'
%%
prog : /* Epsilon */
    | prog decl ';'
    | prog sent ';'
    | prog error ';' { yyerrok; }
;
decl : ID ':' tipo {

```

```

        }
        | ID ',' decl          {
                                // Igual que el anterior
                                }
    ;
tipo   : tipoBase            {
        | RECORD             {
                                registroActual = (nodoRecord
*)malloc(sizeof(nodoRecord));
                                registroActual->enteros = registroActual->cadenas =
NULL;
                                }
        listaDeclBasica
        END RECORD          {
    ;
tipoBase : INTEGER          {
        | STRING            {
    ;
listaDeclBasica : declBasica ';'
                | listaDeclBasica declBasica ';'
                | error ';'      { yyerrok; }
                | listaDeclBasica error ';' { yyerrok; }
                ;
declBasica   : ID ':' tipoBase {

                                }
        | ID ',' declBasica {
                                // Igual que el anterior
                                }
    ;
sent      : PRINT expr      {

                                }
    ;
}

```



```

| ASSIGN IValor '=' expr {
    ;
    }
IValor : ID {
    nodoDato * dato = buscarDato($1);
    if (dato == NULL) {
    }
    if (dato != NULL && dato->tipo=='r') {
    } else {
        if (dato->tipo=='n') $$ptr.ptrInteger=&(dato->valores.valorInteger);
        if (dato->tipo=='c') $$ptr.ptrString=&(dato->valores.valorString);
    }
}
| ID '!' ID {
    ;
    }
}

```

```

expr : IValor      {
    if ($1.tipo=='u')
        $$.tipo='u';
    else {
        $$.tipo = $1.tipo;
        if ($$.tipo == 'n')
            $$.valor.valorInteger = *($1.ptr.ptrInteger);
        else if ($$.tipo == 'c')
            $$.valor.valorString = strdup(*($1.ptr.ptrString));
    }
}
| NUM              {
| CADENA           {
| expr '+' expr    {

```

```

| expr '*' expr    {

```

```

| '(' expr ')'    {
    $$.tipo = $2.tipo;

```

```
        if ($$.tipo=='n') $$valor.valorInteger = $2.valor.valorInteger;
        if ($$.tipo=='c') $$valor.valorString = $2.valor.valorString;
    }

;
%%
#include "errorlib.c"
#include "exdic051.c"
void main(){
    yylineno = 1;
    yyparse();
}
```