Procesadores de Lenguaje. Curso 2001-2002. Junio de 2002. Laboratorio.
No se permiten libros ni apuntes. Tiempo: 1 hora 15 min.

1. (1 ptos) Contesta y justifica la respuesta de las siguientes preguntas:
   - ¿ Admite Bison recursividad a derechas?
   - ¿ Es PCCTS un método de análisis ascendente LR(k) y por eso más potente que el método empleado por Bison?
   - ¿ Cuál es el mecanismo de que dispone Bison para establecer la precedencia de operadores?

2. (1 ptos) Describe como has implementado la eliminación de comentarios tipo C (/* Esto es un comentario tipo C*/) en la práctica 2 con Bison.

3. (1.5 ptos) Dado el siguiente programa en Bison, rellena la partes que faltan e incluye las acciones semánticas correspondientes para que imprima el valor de la expresión y el número de operaciones aritméticas realizadas. Establece las precedencias de operadores habituales.

```
% { /* Ejemplo para una pequeña calculadora que permite trabajar con números
enteros y con las operaciones de suma, resta, producto y división */
#include ...
extern ...
%}
%union { ...  }
%start ...
%token <...> TKN_NUM
...  resto de tokens
%type <...> Calculadora
...  resto de no-terminales
%%
Calculadora :  TKN_ID TKN_ASIGN Exp TKN_PTOCOMA ;
Exp:  TKN_NUM | Exp TKN_MAS Exp | Exp TKN_MENOS Exp | Exp TKN_MULT Exp
| Exp TKN_DIV Exp | TKN_PAA Exp TKN_PAC ;
%%
Código auxiliar
```

4. (1.5 ptos) Indica brevemente las estructuras de datos y funciones usadas para implementar la práctica 1.

5. (2.5 ptos) Supongamos la construcción repetitiva `for` de los lenguajes de programación:

$$StmtFor \rightarrow \textbf{for}(\ StmtAssign\ ;\ Exp\ ;\ StmtAssign\ )\ S*$$

   (i) Indica dónde colocarías los operadores para la construcción del AST con PCCTS. (ii) Implementa la traducción a código intermedio de esta sentencia tal y como lo harías en PCCTS.

6. (2.5 ptos) Corregir los errores existentes en el siguiente fichero de descripción de PCCTS.

```
#token "[\ \t]" <<skip();>>
#token TKN_ID [a-zA-Z] [a-zA-Z0-9]*
#token TKN_NUM [0-9]
#token TKN_INICIO "inicio"
#token TKN_FIN "fin"
#token TKN_PTOCOMA ";"
```

```
#token TKN_ASIG "="
#token TKN_MAS "+"
#token TKN_MENOS "-"
#token TKN_SI "si"
#token TKN_ENTONCES "entonces"
#token TKN_SINO "sino"
#token TKN_IGUAL "=="
#token TKN_MENOR "<"
#token TKN_MAYOR ">"
#token TKN_IMPRIME "imprime"


class MiParser
{
Programa:  TKN_INICIO ^ (Intruccion)* TKN_FIN!;
Instruccion:  (asig ^ | Condicion^ | Imprime^) TKN_PTOCOMA!;
asig :  TKN_ID TKN_ASIG ^ operacion;
operacion:  operacion (TKN_MAS ^ | TKN_MENOS^) operacion | Dato^;
Dato:  TKN_ID | TKN_NUM;
Condicion:  TKN_SI ^ booleano TKN_ENTONCES "{"!  (Intruccion)* "}"!  { TKN_SINO^ "{"!
(Intruccion)* "}"!  }
Booleano:  "(" TKN_ID (TKN_IGUAL^|TKN_MENOR^|TKN_MAYOR^) Dato ")";
Imprime :  TKN_IMPRIME "(" Dato ")"; }
```

Analiza el siguiente código y muestra el árbol que se generaría.

```
inicio
a=2;
si (a<3) entonces
{
      imprime(a);
}
sino
{
      a=4;
};
fin
```