Resolución Parcial

11/07/2023 - Primer Parcial – K2052 - Martes - TN - **SINTAXIS Y SEMÁNTICA DE LENGUAJES**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Nota Final |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |

Marque **la/las opciones correctas** para cada uno de los siguientes enunciados **justificando adecuadamente** cada una de las opciones elegidas. Solo **sumarán puntos** aquellas preguntas donde se haya **marcado correctamente la respuesta** y esté **acompañado de su correcta justificación/desarrollo**. En caso de que una pregunta esté marcada pero no figure su desarrollo, no será válida por más que la/las opciones elegidas sean correctas, y se considerará como falta de cumplimiento de la consigna.

1) Dada la siguiente expresión de C: (( 1 || 0 && -r ) < 0 + s[2] <= -2 != 2 ) - 1 su valor:

a) Su valor depende de r

b) Su valor depende de s

c) Su valor puede ser 0, 1 o -1

d) Ninguna de las otras opciones es correcta

En este ejercicio se debía evaluar paso a paso la expresión dada utilizando la BNF de C para determinar la prioridad y asociatividad de los operadores. A continuación se resalta la operación que se resuelve en cada paso

(( 1 || 0 && -r ) < 0 + s[2] <= -2 != 2 ) - 1
(( 1 || 0 ) < 0 + s[2] <= -2 != 2 ) - 1
( 1 < 0 + s[2] <= -2 != 2 ) - 1
( 1 < s[2] <= -2 != 2 ) - 1

Aquí se podían dar dos situaciones el resolver la operación 1 < s[2]

Caso 1: No se cumple que 1 < s[2]

( 0 <= -2 != 2 ) - 1
( 0 != 2 ) - 1
1 - 1
0

Caso 2: Se cumple que 1 < s[2]
( 1 <= -2 != 2 ) - 1
( 0 != 2 ) - 1
1 - 1
0

Para ambos casos la expresión siempre vale 0. Ahora respondemos cada uno de los puntos.

a) FALSO, el valor no depende de r. Se desprende del desarrollo anterior, queda absorbido en la operación &&.
b) FALSO, el valor tampoco depende de s, analizamos que para ambos casos, termina valiendo 0.
c) FALSO, el valor es cero **siempre**. Decir que ***puede ser***, implica que en algunos casos podría valer 1 o -1.
d) VERDADERO, por no cumplirse ninguna de las otras opciones.

2) En C, el operador &&:

a) Tiene asociatividad de izquierda a derecha

b) Tiene asociatividad de derecha a izquierda

c) Posee mayor precedencia que los operadores relacionales

d) Posee la misma prioridad que el operador not

Este ejercicio se resolvía con la BNF de C donde se puede encontrar en qué producciones aparece el operador &&

***expAnd***: *expIgualdad*

***expAnd*** *&& expIgualdad*

a) VERDADERO, la producción ***expAnd*** del operador && tiene recursividad a izquierda por lo que la asociatividad del operador es de izquierda a derecha.
b) FALSO, por lo mencionado en la justificación anterior, no puede tener ambas asociaciones. Para que este fuese el caso, la recursividad debería ser a derecha.
c) FALSO, posee menor precedencia ya que se encuentra más cerca del axioma, a una menor cantidad de derivaciones.
d) FALSO, el operador not se encuentra más lejos del axioma (linea 31), tiene mayor prioridad. Para tener la misma prioridad deberían poder producirse desde el mismo no terminal.

3) En el proceso de compilación:

| | | | |
|---|---|---|---|
| a) Las etapas de enlazado y ensamblado pueden permutarse | b) La etapa de preprocesamiento puede omitirse | c) Se parte de un programa fuente y se genera un programa ejecutable | d) Ninguna de las otras opciones es correcta |

Este ejercicio se respondía en función del apunte teórico Introducción al proceso de compilación

a) FALSO, las etapas no pueden permutarse, tienen un orden determinado cada una con su entrada y salida. De hecho, no tendría sentido que se permuten dado que la salida del enlazado es un programa ejecutable y no puede ser la entrada de la etapa de ensamblado dado que espera un programa ensamblado para convertirlo en programa objeto.
b) FALSO, ninguna etapa puede omitirse, todas se ejecutan obligatoriamente aunque no tengan ningún efecto. Por ejemplo, aunque no se tuviese que eliminar ningún comentario ni resolver directivas de preprocesamiento, esta etapa se ejecuta de todos modos.
c) VERDADERO, es la definición del proceso de compilación como se define en el apunte.
d) FALSO, dado que c) es correcta.

4) Las gramáticas regulares:

| | | | |
|---|---|---|---|
| a) Generan únicamente lenguajes tipo 3 | b) Pueden generar el lenguaje de las expresiones de C | c) Generan palabras que pueden ser reconocidas por un AFD | d) Pueden tener producciones del tipo S -> aX \| Xa |

Este ejercicio se resolvía aplicando la teoría del volumen 1 sobre gramáticas regulares y lo visto sobre autómatas.

a) VERDADERO, por definición de gramática regular, estas gramáticas generan lenguajes regulares, lo que es equivalente a decir que generan lenguajes tipo 3.
b) FALSO, el lenguaje de las expresiones de C es un lenguaje tipo 2 (LIC - Independiente de contexto), por lo que no es posible generarlo con un GR. Debemos recordar que los LR son un subconjunto de los LIC. Dando un ejemplo práctico, con una GR no es posible generar un lenguaje que contemple el uso de paréntesis que tienen las expresiones.
c) VERDADERO, dado que las GR generan lenguajes regulares (tipo 3) podemos asegurar que para todo lenguaje tipo 3 existe al menos un AFD que reconoce las palabras de dicho lenguaje. Hay una relación equivalente entre GR - AFD - LR.
d) FALSO, no puede tener este tipo de producciones, es Terminal NoTerminal o bien NoTerminal Terminal, pero no puede tener ambas a la vez en la producción.

5) Las gramáticas independientes de contexto:

| | | | |
|---|---|---|---|
| a) Pueden generar el lenguajes de los identificadores de C | b) Pueden generar el lenguaje de sentencias de C | c) No pueden generar lenguajes finitos. | d) No pueden tener gramáticas equivalentes. |

Este ejercicio se resolvía aplicando la teoría del volumen 1 sobre gramáticas independientes de contexto

a) VERDADERO, una GIC genera lenguajes tipo 2 o tipo 3 (abarca un conjunto más grande de lenguajes que puede generar). Dado que el lenguaje de los identificadores es un lenguaje tipo 3, se puede definir

mediante una ERX por lo que podemos asegurar que una GIC puede generar dicho lenguaje, aunque no sea lo más conveniente dado que con un GR podría ser suficiente.

b) VERDADERO, tanto el lenguaje de las sentencias como el de las expresiones de C son LIC, por lo que es suficiente con una GIC para generarlos. En el apunte de hecho está la BNF que muestra cómo se produce el lenguaje de las sentencias.

c) FALSO, puede generar tanto finitos como infinitos. Un ejemplo es el siguiente S -> aaa, genera el lenguaje finito L = {aaa} y como la producción tiene tres terminales seguidos no es regular.

d) FALSO, toda gramática sin importar su tipo tiene gramáticas equivalentes. Con dar un ejemplo es suficiente, por ejemplo la gramática anterior es equivalente a S -> aXa , X -> a dado que generan el mismo lenguaje.

6) Indique cuáles de las siguientes afirmaciones sobre ERX son correctas:

| | | | |
|---|---|---|---|
| a) La ERX [1-7]**+** \| 3{3}? representa un sublenguaje de las constantes octales | b) La ERX 0[x-X]1FA \| 0[xX][89]+ representa un sublenguaje de las constantes hexadecimales | c) La ERX [1-9]*09 NO representa un sublenguaje de las constantes decimales | d) Ninguna de las otras opciones es correcta. |

En este ejercicio se ponía en práctica las ERX vistas en el volumen 1

a) FALSO, dado que la ERX representa el lenguaje que incluye la palabra 3 y 33, y además no garantiza empezar con 0. No hay posibilidad de que genere palabras octales. En todo caso dicha ERX representa un sublenguaje de las constantes decimales.

b) FALSO, el problema con esta ERX es [x-X] dado que contempla un intervalo de caracteres, que de hecho sería nulo porque la "x" está después de la "X" en el código ASCII. Aún siendo al revés [X-x] estaríamos abarcando un montón de caracteres que no cumplirían con empezar con "0X" o "0x", por lo que no pertenecería a ninguna constante entera

c) VERDADERO, dado que que dicha ERX representa el lenguaje que contiene la palabra 09 y dicha palabra no es ningún tipo de constante entera, por lo que podemos decir que NO es un sublenguaje de las constantes decimales.

d) FALSO, dado que la opción c) es correcta.

7) Dado el siguiente fragmento de código C:

```
1       int main (float a, int b) {
2               int a, b = 0xF, p[3] ={0,2.0,-5};
3               a = p[2], a=b++;
4               p[3] = 3;
5               return p[0] && a/p[0];
6               }
```

| | | | |
|---|---|---|---|
| a) Las variables a y b tienen el mismo valor al finalizar la ejecución | b) La ejecución dará error dado que no es posible efectuar la división por 0 | c) La línea 4 es semánticamente equivalente a (*(p)+3) = 3; | d) Los caracteres [] de la línea 2 actúan como caracteres de puntuación |

Este ejercicio apuntaba a analizar en el código C cuestiones relacionadas con operadores de incremento, distinguir entre operadores y caracteres de puntuación, evaluación perezosa (lazy evaluation) y las restricciones sintácticas como el tema de valorL modificable en la asignación.

a) FALSO, la variable b finaliza con un valor distinto al de a dado que tiene un operador de post incremento que se ejecuta luego de asignar el valor de "b" a "a".

b) FALSO, el programa no da error por la división por 0 dado que nunca se llega a evaluar esa parte de la expresión a/p[0] debido a que los operandos de la operación && se evalúan de izquierda a derecha y como p[0] vale 0, la operación de && vale 0 sin necesidad de evaluar el operando que está a la derecha.

c) FALSO, no son equivalentes, y de hecho la expresión de asignación no cumple las restricciones sintácticas dado que en el lado izquierdo no queda un valorL modificable. *(p)+3 es el valor de "p" más 3 (una constante) y no se le puede asignar un valor a una constante.

d) VERDADERO, en la línea 2 tenemos una declaración de un array, por lo que los [] actúan como caracter de puntuación dado que no están cumpliendo el rol de operador de acceso.

8) Dado el siguiente fragmento de código C:

```
1       int main (float a, int b) {
2               int a='A', p[3] = {0,1,2};
3               a = p;
4               return {b};
5               }
```

| | | | |
|---|---|---|---|
| a) Posee dos errores sintácticos | b) Es léxicamente correcto y posee un total de 40 lexemas | c) Posee dos errores semánticos | d) Posee un error sintáctico y un error semántico |

Este ejercicio tenía algunas posibilidades en su resolución por haber una cuestión semántica que puede considerarse como warning o como error.

Errores semánticos posibles:

- Error en línea 2 - Doble declaración de la variable "a". Dado que es un parámetro de entrada, no se puede declarar dentro de la función una variable con el mismo nombre. E*ste error semántico se debía encontrar obligatoriamente para considerar correcto el ejercicio.*
- Error en línea 3 - Error de tipos: Dado que la variable p es de tipo puntero *int, y la variable a es de tiempo int. Este error puede considerarse como un warning de tipos dado que el programa ejecutará en algunos compiladores resolviendo la conversión de *int a int automáticamente. *Este segundo error podía omitirse y el ejercicio se consideraba correcto de todos modos.*

Errores sintácticos:

- Error en línea 4, luego de return debe venir una expresión. También podría considerarse como que luego de "b" falta un ; debido a que es una sentencia. *De ambas formas se considera como error sintáctico y es obligatorio detectarlo para poder considerar correcto el ejercicio.*

a) FALSO, posee un solo error sintáctico
b) FALSO, en total hay 38 lexemas.
c) VERDADERO, esta opción se considera válida si indicaban ambos errores semánticos descritos anteriormente. Caso contrario no debería marcarse, debe haber coherencia entre la justificación y la opción elegida.
d) VERDADERO, esta opción se considera válida en caso de que solo hayan marcado uno de los errores semánticos. Misma consideración que para el punto c)

9) Dadas las siguientes reglas de flex y la cadena de entrada, :

Entrada yyin: ''000871G165Fazt850C05ag''  lexemas reconocidos: 9  yyout = G0

Reglas de Flex:

| | | |
|---|---|---|
| Regla 1 | [1-9]+ | ; |
| Regla 2 | [a-z]+ | contador += 1; |
| Regla 3 | [0-8]{3,5} | acumulador = acumulador + atoi(yytext); |
| Regla 4 | [A-Fa-g] | acumulador += 2; |

Considerando que las variables acumulador y contador están inicializadas en 0. Al finalizar la ejecución del análisis léxico por un scanner que aplicar las reglas definidas previamente sobre la entrada yyin.

| | | | |
|---|---|---|---|
| a) La salida yyout es ''G'' | b) Se reconocen en total 10 lexemas. | c) Se emparejó tres veces por la regla 1 | d) Los valores de las variables acumulador y contador son 941 y 2 respectivamente |

Para resolver este ejercicio se debía procesar la cadena de entrada yyin siguiendo las reglas de flex:

e) FALSO, la salida es la cadena "G0"
f) FALSO, se reconocen en total 9 lexemas (cantidad total de veces que empareja por una regla.
g) VERDADERo, en total emparejo tres veces, primero con "1", luego con "165" y por último con "5"
h) VERDADERO, se desprende de la resolución, dado que entra dos veces por la regla 2, la variable contador termina valiendo 2. Y acumulador sale de 87 + 2 + 850 + 2 =941 entrando por las reglas 3 y 4.

10) Cuando trabajamos con autómatas finitos, podemos asegurar que:

| | | | |
|---|---|---|---|
| a) La intersección de dos AFD siempre da como resultado un AFD con un mayor número de estados alcanzables. | b) Para hallar el complemento de un AFD se requiere que el estado inicial no sea final. | c) Todo AFN posee un AFD equivalentes | d) Es posible implementar computacionalmente un AFN |

Este ejercicio se resolvía aplicando la teoría de autómatas incluyendo las operaciones de intersección y complemento

a) FALSO, la intersección de dos AFD no siempre tiene un mayor número de estados alcanzables. Por ejemplo si hacemos la intersección de un AFD que reconoce un lenguaje con el AFD que reconoce su complemento, nos dará como resultado un AFD con ningún estado alcanzable a excepción del estado inicial desde donde comienza.
b) FALSO, esa condición no es necesaria. En un AFD si el estado inicial además es final, eso indica que al menos reconoce la palabra nula, lo cual no es un impedimento para hallar el complemento de dicho AFD. El procedimiento no cambia por esta condición, lo que va a suceder es que en el AFD complemento el estado inicial no será final, es decir que podemos garantizar que en el lenguaje complemento no estará la palabra nula.
c) VERDADERO, todo AFN tiene un AFD equivalente. Hasta el momento lo vimos teóricamente sin demostrarlo pero mencionamos que esto se hace a través del algoritmo de Clausura.
d) FALSO, los AFN no pueden implementar computacionalmente dado que no cumplen la unicidad en la función de transición y para implementarlos deben ser determinísticos para que para cada par (estado, caracter) tenga un único estado posible al que transitar.