SINTAXIS Y SEMÁNTICA DE LOS LENGUAJES

TRABAJO PRÁCTICO INTEGRADOR

Dígrafos y trígrafos

Hedderwick, Federico   
134-960-0 dic-2015

Dada la siguiente secuencia de caracteres, es decir una Unidad de Traducción:

int printf(const char\*,...);

int main(void){

int \_[:>=<%-!.0,};

printf("%d%d",sizeof \_-sizeof \_[0],sizeof(char)+0[\_]);

}

Se procede a realizarle un análisis léxico, sintáctico y semántico, comenzando por el léxico. Previamente a buscar los lexemas y analizarlos para formar tokens, se dara una breve introducción al proceso:

Un programa escrito en lenguaje C se divide en tokens separados por espacios blancos y simbolos de puntuación, que llamaremos puntuadores para facilitar la escritura de este informe.  
Un lexema puede ser un caracter o una secuencia de ellos, que forman una unidad. Al darle un significado, pasa a ser un token. El compilador primero procesará la Unidad de Traducción en esta secuencia de tokens y puntuadores, en un análisis léxico.  
Luego, llevará a cabo un análisis sintáctico con dicha secuencia de tokens y puntuadores.  
Por último llevará a cabo un análisis semántico para poder crear el código objeto que el linker unirá para formar el archivo ejecutable final.

Ahora sí, se pasa al análisis léxico:

**ANALISIS LEXICO**

Los digrafos y trigrafos son secuencias de dos y tres caracteres respectivamente, que son tenidas en cuenta como si fuesen un solo caracter. La idea detrás de esto es poder generar caracteres necesarios en caso de que el teclado que se esté utilizando no disponga de ellos.

Los trígrafos son procesados por el preprocesador, a diferencia de los dígrafos que son procesados durante el tokenizado. Estos deben representar un token entero por sí mismos. Si un dígrafo se encuentra dentro de otro token, como por ejemplo una cadena de texto o un carácter, la secuencia de caracteres no será dígrafo, por lo que no será reemplazada.

Los tokens pueden ser de cinco tipos:

-**Literales**: secuencia de caracteres, delimitado por espacios o caracteres especiales, sean operadores o signos de puntuacion.

-**Palabras** **clave**: son tokens predefinidos por el lenguaje, como por ejemplo las palabras "int", "void", "const".

-**Identificadores**: son palabras que identifican las variables y funciones. Pueden tener letras, números y el guión bajo, con la única restricción de que un número no sea el primer caracter.

-**Puntuadores**: sirven para separar, marcar, y ordenar tokens, como por ejemplo paréntesis, llaves, corchetes, comsa, etc.

-**Operadores**: sirven para justamente operar los datos. Los hay de un caracter (como ~ ! @ % ^ & \* - + = | / : ? < > ,), de dos (++ -- << >> <= += -= \*= /= == |= %= &= ^= || && !=) o de tres caracteres (<<= >>=)

Tras esta introducción, se pasará al análisis léxico propiamente dicho de la unidad de traducción:

Lexemas (unidades lexicas) en secuencia:

int printf ( const char \* , ... ) ; int main ( void ) { int \_ [ : > = < % - ! . 0 , } ; printf ( " %d%d , sizeof \_ - sizeof \_ [ 0 ] , sizeof ( char ) + 0 [ \_ ] ) ; }

Al darles un significado, se pueden especificar como tokens:

int palabra reservada  
printf identificador (de función)  
() operador (invocación a función)  
const palabra reservada (modificador)  
char palabra reservada  
\* operador (indicación de puntero)  
, puntuador (separador)  
... puntuador (indicación de cantidad variable)  
; puntuador (terminador de sentencia)   
int reservada  
main identificador (de función)  
() operador (invocación a función)  
void palabra reservada  
{} puntuador (bloque de código)  
int palabra reservada  
\_ identificador (nombre del array)  
[] operador (offset para declarar array). El de cierre es un dígrafo :> reemplazado.  
= operador (asignación)  
<% } dígrafo reemplazable por {, son puntuadores (conjunto que inicializa el array)  
- operador (unario: signo)  
! operador (negación)  
.0 literal (valor numérico de punto flotante)  
, puntuador (separador de elementos en conjunto)  
; puntuador (terminador de sentencia)  
printf identificador (de función)  
( ) operador (invocación a función)  
" puntuador (inicio de literal cadena)  
%d%d literal (valores numéricos a reemplazar por los argumentos de la función)  
, puntuador (separador de expresiones)  
sizeof palabra reservada  
\_ identificador (nombre del array)  
- operador (resta)  
sizeof palabra reservada  
\_ identificador (nombre del array)  
[] operador (offset de array)  
0 literal (valor numérico entero)  
, puntuador (separador de expresiones)  
sizeof palabra reservada  
() operador (invocación a función)  
char palabra reservada  
+ operador (suma)  
0 literal (valor numérico entero (tomando lugar de base del array))  
[] operador (offset de array)  
\_ identificador (nombre del array (tomando lugar del offset))  
; puntuador (finalizador de sentencia)

Como no dio ningún error léxico, es decir no hay un desbalance entre aperturas y cierres, ni lexemas extraños al lenguaje, la Unidad de Traducción, es decir la secuencia de caracteres, es léxicamente correcta. Se procede entonces al análisis sintáctico de la UT:

**ANÁLISIS SINTÁCTICO:**

El análisis sintáctico se esquematiza en un árbol de derivación. El mismo, por su naturaleza, se encuentra en un archivo separado.

Este análisis termina de forma correcta, no se reciben tokens en un orden inesperado ni en posiciones que no corresponden a las reglas del lenguaje.

Una vez asegurada la integridad sintáctica, se realiza el análisis semántico:

**ANALISIS SEMANTICO:**

Nuevamente se pide analizar si la UT es léxicamente correcta, lo cual parece ser un error del enunciado del práctico, donde probablemente se quiso decir “semánticamente” pero se dijo “léxicamente”. Este análisis semántico deberá realizarse con el fin de detectar irregularidades tales como asignaciones prohibidas, que son sintácticamente correctas pero inválidas por las reglas del lenguaje C. En este caso, la UT es semánticamente correcta, pues no viola ninguna restricción semántica.

El programa lo que hace es:

int printf(const char\*,...);

Declara una función “printf”, que devuelve un int y recibe como parámetros un puntero a un array constante de char, y puede recibir más parámetros en un número indefinido que inicia en cero.

int main(void){

Define una función “main”, que devuelve un int y recibe como parámetro un void, es decir, nada. La llave del final indica que será definida a continuación, dentro del bloque que terminará con una llave de cierre.

int \_[:>=<%-!.0,};

Tras convertir los dígrafos quedaría:

int \_[]={-!.0,};

Declara una variable llamada “\_”, que al tener a continuación el par [], indica ser un array. Ese array es de int, y se inicializa por medio del operador igual. Al tener un bloque entre llaves a continuación, dentro estarán los valores de inicialización, que en este caso serán “el negativo de la negación del literal “.0””, donde ese punto hace que el literal numérico sea flotante. La primera operación, negación, hará que ese literal flotante pase a valer 1, convertido automáticamente a un entero por efecto de la negación. A este valor se le cambia el signo, pasando a valer -1. La coma a continuación es un separador de parámetros, omisible en este caso por ser el último valor. La llave cierra el bloque y el “;” transforma la expresión en sentencia.

printf("%d%d",sizeof \_-sizeof \_[0],sizeof(char)+0[\_]);

Mediante el identificador “printf”, referencia a dicha función, y al estar seguido de paréntesis, estos actúan como operador invocación. Dentro de ellos irán los parámetros que serán enviados a la función. Primero y siguiendo lo que informa el prototipo, debe ir un literal cadena. Este está compuesto por dos inyecciones en formato entero de parámetros, por lo que se espera que lleguen dos parámetros más (el % informa una inyección de parámetro y la “d” que será tratada como un entero). El segundo parámetro está formado por la palabra reservada sizeof seguida, espacio de por medio, del identificador del array mencionado anteriormente. Esto dará como resultado el tamaño del array. A ese valor se le restará (por el operador “-“) el resultado de evaluar “sizeof \_[0]”, es decir, el tamaño del elemento inicial del array “\_”. Por último, el tercer parámetro será el resultado de la suma entre la evaluación de “sizeof(char)” (que da como resultado 1, pues es el tamaño del tipo de dato “char”), y de la operación “offset de array”, donde curiosa pero válidamente se intercambian la referencia al array con el desplazamiento. Esto es válido pues desplazarse, por ejemplo 5 bytes desde un valor ejemplo 3, es lo mismo que desplazarse desde 3 en 5 posiciones. De cualquier manera, la evaluación de esta expresión será el primer elemento del array. Puesto que el mismo era un flotante,

Finalmente llegará el paréntesis de cierre que es parte del operador invocación, luego el punto y coma que hará que la expresión sea una sentencia, y a continuación la llave de cierre del bloque “definición de la función main”.

La salida por stdout es “00”, debido a que son dos enteros los que se imprimen. El primero da cero porque es el resultado de “sizeof \_-sizeof \_[0]”, que son iguales: en este caso el array se inicializó en su creación con -1, que es un entero. El minuendo dará entonces el tamaño de palabra del procesador. El sustraendo dará también ese tamaño, pues está pidiendo el tamaño del primer elemento, que es justamente ese -1.

En el caso del segundo entero a imprimir, también dará cero, pues “sizeof(char)+0[\_]” tendrá en su primer sumando el valor 1, y en el segundo, el -1 . Esto último debido a que el primer y único elemento del array es el entero -1, como se mencionó al analizar la incicialización del array; el poder conseguir el primer elemento intercambiando offset con nombre de array fue explicado en el análisis semántico.