Procesadores del Lenguaje

Curso 2020-2021

Práctica Final – Primera Parte.

Traductor de Expresiones a un Lenguaje en Notación Prefija

Esta primera parte de la práctica consiste en aumentar el traductor de la sesión previa.

Seguiremos con el esquema de traducción que nos facilita propagar hacia arriba en la gramática las traducciones que se van realizando. Ahora se ampliará la gramática para permitir la traducción de fragmentos de código C. La salida seguirá siendo en notación prefija, la que corresponde a una variante del Lenguaje Lisp.

Trabajo a realizar:

Para esta aproximación se pide la traducción de expresiones aritméticas y de asignaciones a variables y algunas sentencias que corresponden al Lenguaje C a notación prefija que serán evaluadas por un intérprete de Lisp.

1. Leed el enunciado completo antes de comenzar.
2. Estudiad el código `trad2.y` proporcionado y explicado en las páginas siguientes.
3. Incluid **la gramática** de vuestra práctica anterior `trad1.y` en `trad2.y` . En principio debería ser sólo la sección gramatical (producciones y semántica). No modifiquéis `yylex()`.
4. Desarrollad los puntos indicados en las Especificaciones hasta donde os de tiempo. En futuras sesiones continuaremos con ellas.
5. Podéis evaluar los resultados de traducción usando el intérprete online `https://rextester.com/l/common_lisp_online_compiler`. Para ello podéis:
   a. editar un fichero (por ejemplo, `prueba.txt`) con una serie de expresiones
   b. ejecutar `cat prueba.txt | ./trad2`
   c. copiar la salida y pegarla en la ventana del intérprete
   d. con F8 se compila y ejecuta.

Entrega:

Subid el fichero `trad2.y` con el trabajo realizado hasta el momento.

Incluid en la cabecera del fichero dos líneas de comentario iniciales. La primera con vuestros nombres y número de grupo. Y la segunda con los correos electrónicos (separados con un espacio).

Entregad también un documento llamado `trad2.pdf` con una breve explicación del trabajo realizado.

Después de hacer la entrega, descargad vuestro fichero y comprobad que funciona correctamente y que cumple con las indicaciones.

Código auxiliar propuesto para la Práctica Final

Se proporciona a continuación un código de partida para resolver diversos aspectos relacionados con la práctica. Es importante que entendáis el funcionamiento.

Se proporcionan las estructuras de datos y primitivas necesarias, en concreto, las relacionadas con las palabras reservadas y el analizador lexicográfico dedicado.

El bloque inicial de *yylex* realiza diversas funciones relacionadas con separadores y comentarios **que se comentarán de más adelante y por separado**.

```
int yylex ()
{
    int i ;
    unsigned char c ;
    unsigned char cc ;
    char ops_expandibles [] = "!<=>|%&+-/*" ;
    char cadena [256] ;
    t_reservada *simbolo ;

    do {
         c = getchar () ;

                if (c == '#') {
                // Ignora las lineas que empiezan por #  (#define, #include)
                        do {
                        //       OJO que puede funcionar mal si una linea
contiene #
                         c = getchar () ;
                        } while (c != '\n') ;
                }

                if (c == '/') {  // Si la linea contiene un / puede ser
inicio de comentario
                        cc = getchar () ;
                        if (cc != '/') {   // Si el siguiente char es /  es
un comentario, pero...
                                ungetc (cc, stdin) ;
                 } else {
                     c = getchar () ;     // ...
                     if (c == '@') {      // Si es la secuencia //@  ==>
transcribimos la linea
                       do {            // Se trata de codigo inline (Codigo
embebido en C)
                           c = getchar () ;
                           putchar (c) ;
                       } while (c != '\n') ;
                     } else {            // ==> comentario, ignorar la linea
                       while (c != '\n') {
                           c = getchar () ;
                       }
                     }
                 }
                }

                if (c == '\n')
                 n_linea++ ;

    } while (c == ' ' || c == '\n' || c == 10 || c == 13 || c == '\t') ;. . .
}
. . .
```

Un aspecto que se reforma en `yylex` es el tratamiento de los separadores. Antes se consideraba como tal el espacio en blanco. Ahora lo ampliamos a los tabuladores y los saltos de línea (\n y \r). En este caso ya no nos interesa terminar una expresión con un <intro>, sino que buscamos separar una sentencia del lenguaje de otra. Lo haremos con el carácter '`;`' correctamente identificado como literal en la gramática del parser. Para ignorar los separadores mencionados usaremos el siguiente código:

```
int yylex ()
{

    do {
. . .
    } while (c == ' ' || c == '\n' || c == '\r' || c == '\t') ;
. . .
```

El siguiente fragmento sirve para detectar cadenas literales en la entrada. Cuando el carácter leído es " , el *scanner* entrará en un bucle para recopilar la secuencia de caracteres hasta que aparezca otro ". Dicha secuencia se almacena en cadena, y se devuelve como *Token* STRING. Su uso será en funciones tipo *printf*.

```
    if (c == '\"') {
         i = 0 ;
         do {
             c = getchar () ;
             cadena [i++] = c ;
         } while (c != '\"' && i < 255) ;
         if (i == 256) {
              printf ("AVISO: string con mas de 255 caracteres en linea %d\n",
n_linea) ;
         }       // habria que leer hasta el siguiente " , pero, y si falta?
         cadena [--i] = '\0' ;
         yylval.cadena = genera_cadena (cadena) ;
         return (STRING) ;
    }
```

Los números se tratan en el siguiente fragmento. Aunque se detecte un punto decimal, se tratarán como números enteros.

```
    if (c == '.' || (c >= '0' && c <= '9')) {
         ungetc (c, stdin) ;
         scanf ("%d", &yylval.valor) ;
//         printf ("\nDEV: NUMERO %d\n", yylval.valor) ;      // PARA DEPURAR
         return NUMERO ;
    }
```

Aquí vamos a asumir que las *palabras reservadas* de un lenguaje son todas aquellas de las que no podremos disponer para definir nuevas variables y funciones, por estár ya definidas. En el caso del lenguaje C, éstas serán `if, then, else, for, while`, etc. pero también podemos incluir `main`, o incluso funciones de uso habitual como `printf, puts`, o casos como operadores compuestos como `<=` , etc. que no se pueden resolver

bien mediante literales. El esquema para detectar estas palabras reservadas consiste en disponer de una tabla que las identifique y las relacione con su correspondiente *token* del parser. El analizador lexicográfico deberá consultar esta tabla cada vez que lea una secuencia de caracteres de tipo identificador.

```
    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
          i = 0 ;
          while (((c >= 'A' && c <= 'Z')   || (c >= 'a' && c <= 'z') ||
                    (c >= '0' && c <= '9')  || c == '_') && i < 255) {
              cadena [i++] = tolower (c)   ;      // OJO PASO TODO A MINUSCULA
              c = getchar () ;
          }
          cadena [i] = '\0' ;                     // Terminacion de cadena
          ungetc (c, stdin) ;                     // Devolver char sobrante
          simbolo = busca_pal_reservada (cadena) ;
          if (simbolo == NULL) {   // no es palabra reservada -> identificador
              yylval.cadena = genera_cadena (cadena) ;
//            printf ("\nDEV: IDENTIFICADOR %s\n", cadena);   // PARA DEPURAR
              return (IDENTIFICADOR ) ;
          } else {
              yylval.cadena = NULL ;
//            printf ("\nDEV: OTRO %s\n", cadena) ;           // PARA DEPURAR
              return (simbolo->token) ;
          }
    }
```

Si una secuencia de caracteres empieza por un carácter alfabético, será recopilada con un bucle mientras los posteriores caracteres sean alfanuméricos o _. Prestad atención al detalle de que todos los caracteres alfabéticos se pasan a minúscula con la función `tolower`. Una vez recopilada y terminada la secuencia, se comprueba si la secuencia leída corresponde a una *palabra reservada* o no mediante `busca_pal_reservada`. En el caso de no ser encontrada, se devuelve con el *token* `IDENTIFICADOR`. Previamente se copia a un fragmento de memoria dinámica con la función `genera_cadena` para asignarla a `yylval.cadena`. En el caso de ser identificada en la tabla, se devolverá el *token* correspondiente, con `yylval.cadena` apuntando a `NULL`.

Un caso especial son los operadores lógicos compuestos, como `==`, `&&`, que se tratarán como palabras reservadas. Cuando se detecte el inicio posible de uno de estos (cualquier carácter contenido en `ops_expandibles`) se combinará con el siguiente símbolo para determinar si en conjunto están definidos como palabra reservada. En caso contrario, se retornará el segundo carácter a la entrada para devolver el primero como literal.

```
    char ops_expandibles [] = "!<=>|%&+-/*" ;
. . .
    if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
         cc = getchar () ;
         sprintf (cadena, "%c%c", (char) c, (char) cc) ;
         simbolo = busca_pal_reservada (cadena) ;
         if (simbolo == NULL) {
              ungetc (cc, stdin) ;
              yylval.cadena = NULL ;
              return (c) ;
         } else {
              yylval.cadena = genera_cadena (cadena) ; // aunque no se use
              return (simbolo->token) ;
         }
    }
```

La última opción consiste devolver como literal el carácter que no ha sido filtrado anteriormente.

```
//    printf ("\nDEV: LITERAL %d #%c#\n", (int) c, c) ;      // PARA DEPURAR
    if (c == EOF || c == 255 || c == 26) {
//         printf ("tEOF ") ;                                 // PARA DEPURAR
         return (0) ;
    }

    return c ;
}
```

A continuación, vemos las definiciones respecto a las *palabras reservadas* que incluyen tipos y estructuras de datos. En la tabla de *palabras reservadas* se asocia cada una de ellas (ojo, que tendrán que estar siempre en minúscula) con su correspondiente *token*.

```
typedef struct s_pal_reservadas { // para las palabras reservadas de C
    char *nombre ;
    int token ;
} t_reservada ;

t_reservada pal_reservadas [] = { // define las palabras reservadas y los
    "main",         MAIN,           // y los token asociados
    "int",          INTEGER,
    NULL,           0               // para marcar el fin de la tabla
} ;

t_reservada *busca_pal_reservada (char *nombre_simbolo)
{                                   // Busca n_s en la tabla de pal. res.
                                    // y devuelve puntero a registro (simbolo)
    int i ;
    t_reservada *sim ;

    i = 0 ;
    sim = pal_reservadas ;
    while (sim [i].nombre != NULL) {
           if (strcmp (sim [i].nombre, nombre_simbolo) == 0) {
                                        // strcmp(a, b) devuelve == 0 si a==b
                return &(sim [i]) ;
             }
           i++ ;
    }

    return NULL ;
}
```

La función `busca_pal_reservada` devuelve un puntero al registro de la tabla si se encuentra la palabra, o `NULL` en caso contrario. Inicialmente se definen dos *palabras reservadas*: `int` y `main`.

Otras funciones ya conocidas son para asignación y gestión básica de memoria dinámica `mi_malloc` y `genera_cadena`.

```
char *mi_malloc (int nbytes)       // reserva n bytes de memoria dinamica
{
    char *p ;
    static long int nb = 0;        // sirven para contabilizar la memoria
    static int nv = 0 ;            // solicitada en total

    p = malloc (nbytes) ;
    if (p == NULL) {
         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
         exit (0) ;
    }
    nb += (long) nbytes ;
    nv++ ;
    return p ;
}
. . .
char *genera_cadena (char *nombre)     // copia el argumento a un
{                                      // string en memoria dinamica
      char *p ;
      int l ;

      l = strlen (nombre)+1 ;
      p = (char *) mi_malloc (l) ;
      strcpy (p, nombre) ;
      return p ;
}
```

Ambas existen ya como a `malloc` y `strdup` pero se redefinen aquí para que se entienda su funcionamiento y para añadir alguna funcionalidad.

En la primera sección del *parser* se añaden ficheros de cabecera para declarar funciones necesarias de comparación y copia de cadenas (`string.h`) y para `exit` (`stdlib.h`).

```
%{                            // SECCION 1 Declaraciones de C-Yacc
#include <stdio.h>
#include <string.h>             // declaraciones para cadenas
#include <stdlib.h>             // declaraciones para exit ()
#define FF fflush(stdout);      // para forzar la impresion inmediata
%}
```

En cuanto al tipo de pila, puesto que ya no necesitamos evaluar variables, ni estas serán de un solo carácter, el tipo índice no nos sirve. Lo sustituimos por el tipo char* (tipo string), que contendrá la secuencia de caracteres del nombre de cada variable.:

```
%union {                        // El tipo de la pila tiene caracter dual
      int valor ;               // - valor numerico de un NUMERO
      char *cadena ;            // - para pasar los nombres de IDENTIFICADORES
}
%token <valor> NUMERO           // Todos los token tienen un tipo para la pila
%token <cadena> IDENTIFICADOR // Identificador=variable
%token <cadena> INTEGER         // identifica la definicion de un entero
%token <cadena> MAIN            // identifica el comienzo del proc. main
```

Se han definido también los *token* utilizados. Conviene indicar que el hecho de que `INTEGER` y `MAIN` son de tipo cadena es intrascendente. No nos hace falta saber la secuencia de caracteres original para estos (`int` o `main`) porque los mismos *token* ya contienen la información necesaria.

Especificaciones

Proponemos una secuencia de pasos para desarrollar esta práctica. Se recomienda abordar cada uno de los pasos de forma secuencial. Pero en caso de complicarse alguno, se puede pasar al siguiente.

1. Retomando la sentencia simplificada para imprimir, el *scanner* actual procesa el símbolo `#` de forma especial, así que puede que no funcione. Cambiad el símbolo de impresión `#` por el símbolo `$`. Incluid también los paréntesis para englobar la función `$ (<exp>) ;` debe traducirse a `(print <exp>)` [1].

2. Adaptad la gramática para que traduzca la impresión con múltiples parámetros. La entrada `$ (<exp1>, <exp2>, <exp3>) ;` debe traducirse a `(print <exp1>) (print <exp2>) (print <exp3>)`, en el mismo orden original.

3. Incluid la definición de variables globales en la gramática. La definición en C será `int <id> ;` [2] que debe traducirse a `(setq <id> 0)`. Es necesario incluir un segundo parámetro en Lisp para inicializar las variables. En el caso de la inicialización más simple en C, este valor será 0 por omisión.

4. Ampliad la gramática para contemplar la definición de una variable con asignación incluida: `int <id> = <cte> ;` que debe ser traducido a `(setq <id> <cte>)`. Utilizamos aquí `<cte>` para representar un valor numérico constante. No consideraremos por ahora expresiones en las declaraciones.

5. En C `main` es el procedimiento/función principal, Debe tener una palabra reservada en la tabla correspondiente, y debe enlazar con el *Token* `Main`. Ampliad la gramática para reconocer la función `main`. Debe permitir la inclusión de sentencias. Prestad atención al siguiente punto. Incluimos un ejemplo de traducción:

C	Lisp

`int a ;`
`main () {`
`  $ (a + 1) ;`
`}`
	`(setq a 0)`
	`(defun main ()`
	`  (print (+ a 1))`
	`)`

	`(main) ; Para ejecutar el programa`

6. Considerad que la estructura de un programa en C debe ser:
   `<Decl Variables> <Def Funciones>.` En teoría debería poder intercalarse ambos tipos de definiciones, pero eso puede producir conflictos. Por ello seguiremos una estructura fija. Las sentencias que define la gramática sólo deben aparecer dentro del cuerpo de una función. Revisad la estructura de vuestra gramática. Debe estar diseñada de forma muy cuidadosa y estructurada, intentando que sea lo más jerárquica posible.

[1] Más adelante lo sustituiremos por la sentencia correspondiente a 'printf(..)' de C
[2] A partir de este punto comenzamos a traducir de C a Lisp

7. Dentro del cuerpo de las funciones pueden declararse variables, que tanto en C como en Lisp serán variables locales. Se emplea la misma traducción de las globales, pero en este caso deben generarse dentro del cuerpo de la función:

C	Lisp

`int a ;`
`main () {`
`  int a = 4 ;`
`  $ (a + 1) ;`
`}`
	`(setq a 0)`
	`(defun main ()`
	`  (setq a 4)`
	`  (print (+ a 1))`
	`)`

	`(main) ; Para ejecutar el programa`

   Hay que tener cuidado a la hora de incluir la definición de variables locales en la gramática, para evitar conflictos con la definición de las globales. Esta traducci

8. Ampliad la gramática para contemplar la definición múltiple de variables con asignaciones opcionales: **`int <id1> = 3, <id2>, …, <idk> = 1 ;`** que debe ser traducido a una secuencia de definiciones individuales

   `(setq <id1> 3) (setq <id2> 0) … (setq <idk> 1)` . Intentad que el orden de impresión de las variables corresponda al de la declaración

9. Continúa…En la siguiente parte veremos la definición y uso de:
   a. Estructuras de control.
   b. Vectores, Matrices.
   c. Funciones.

A la hora de evaluar esta práctica tened en cuenta que la dificultad será incremental conforme avancen los puntos planteados. Los primeros se pueden resolver en clase por lo que también contarán menos. En las últimas sesiones se valorará vuestra autonomía a la hora de completar los puntos propuestos.

No se recomienda en ningún caso dejar la práctica para última hora. Las sesiones de prácticas están pensadas para dar soporte con los números errores que surgirán al usar *bison*.