Procesadores del Lenguaje

Curso 2024-2025

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. Descargad el código **`trad1e.y`** y renombradlo a **`trad1.y`**. Estudiad el código proporcionado y explicado en las páginas siguientes.
   a. El resumen de cambios de la próxima página.
   b. La estructura de pila y sus atributos
   c. Los nuevos *token*.
   d. Las funciones de código C disponibles y comentadas más adelante
   e. La gramática que soluciona la aproximación previa.
3. Desarrollad los puntos indicados en las Especificaciones hasta donde os de tiempo. En futuras sesiones continuaremos con ellas.
4. 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 | ./trad1`**
   c. copiar la salida y pegarla en la ventana del intérprete
   d. con F8 se compila y ejecuta.

Entrega:

Subid el fichero **`trad1.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 **`trad1.pdf`** con una breve explicación del trabajo realizado.

Antes y después de hacer la entrega, comprobad que funciona correctamente y que cumple con las indicaciones.

Pasos de integración / RESUMEN DE CAMBIOS

- Se han eliminado las directivas **`%type`** para asegurar el uso de atributos explícitos. No se recomienda usar la versión implícita salvo que tengáis soltura a la hora de depurar los errores que pueden provocar. En los exámenes se os pedirá usar la forma explícita.
- Se sustituye la estructura **`%union`** por un **`struct`**.
- Los atributos de la estructura se renombran como **`valor`**⇒**`value`** y **`cadena`**⇒**`code`**. Se añade **`node`** para posible uso de AST en el futuro.
- Se ha cambiado el *token* `VARIABLE` por `IDENTIF`. A partir de ahora, los nombres de variables podrán servir también cómo nombre de vectores y funciones, de ahí el cambio a `IDENTIF`. Ha pasado de usar el atributo **`indice`** (**`%c`**) a **`code`** (**`%s`**).
- Se han añadido nuevos *token*.
- Se añaden las estructuras y funciones para gestionar los AST binarios.
- Se renombran funciones de C y variables a su versión en inglés:
    - **`genera_cadena`** ⇒ **`gen_code`**
    - **`busca_palabra_reservada`** ⇒ **`search_keyword`**
    - etc.
- The grammar from the second approach has been incorporated:
- **`yylex`** ya no proporciona saltos de línea '\n', se sustituyen por ';' para marcar el fin de cada sentencia.

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 más adelante y por separado**.

```
int yylex ()
{
    int i ;
    unsigned char c ;
    unsigned char cc ;
    char expandable_ops [] = "!<=>|%/&+-*" ;
    char temp_str [256] ;
    t_keyword *symbol ;

    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 / posible 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 //@  ==> la transcribimos
                        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_line++ ;

    } 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 (*strings*) 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 **`temp_str`**, y se devuelve como *Token* `STRING`. Su uso será en funciones tipo *printf* y *puts*.

```
    if (c == '\"') {
        i = 0 ;
        do {
            c = getchar () ;
            temp_str [i++] = c ;
        } while (c != '\"' && i < 255) ;
        if (i == 256) {
            printf ("WARNING: string with more than 255 characters in line %d\n",
n_line) ;
        }
        temp_str [--i] = '\0' ;
        yylval.code = gen_code (temp_str) ;
        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.value) ;
        return NUMBER ;
    }
```

Aquí vamos a asumir que las *palabras reservadas* (*keywords*) 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 procedimiento 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) {
            temp_str [i++] = tolower (c) ;
            c = getchar () ;
        }
        temp_str [i] = '\0' ;
        ungetc (c, stdin) ;

        yylval.code = gen_code (temp_str) ;
        symbol = search_keyword (yylval.code) ;
        if (symbol == NULL) {    // no es palabra reservada -> identificador antes
variable
            return (IDENTIF) ;
        } else {
            return (symbol->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 la función **`search_keyword`**. En el caso de no ser encontrada, se devuelve con el *token* `IDENTIF`. Previamente se copia a un fragmento de memoria dinámica con la función **`gen_code`** para asignarla a **`yylval.code`**. En el caso de ser identificada en la tabla, se devolverá el *token* correspondiente.

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 **`expandable_ops`**) 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 expandable_ops [] = "!<=>|%/&+-*" ;
. . .
    if (strchr (expandable_ops, c) != NULL) { // busca c en operadores expandibles
        cc = getchar () ;
        sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
        symbol = search_keyword (temp_str) ;
        if (symbol == NULL) {
            ungetc (cc, stdin) ;
            yylval.code = NULL ;
            return (c) ;
        } else {
            yylval.code = gen_code (temp_str) ; // aunque no se use
            return (symbol->token) ;
        }
    }
```

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

```
    if (c == EOF || c == 255 || c == 26) {
        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 `keywords` (ojo, que tendrán que estar siempre en minúscula) se asocia cada una de ellas con su correspondiente *token*.

```
typedef struct s_keyword { // para las palabras reservadas de C
    char *name ;
    int token ;
} t_keyword ;

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

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

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

    return NULL ;
}
```

La función **search_keyword** 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 `my_malloc` y `gen_code`.

```
char *my_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 memoria left for additional %d bytes\n", nbytes) ;
        fprintf (stderr, "%ld bytes reserved in %d calls\n", nb, nv) ;
        exit (0) ;
    }
    nb += (long) nbytes ;
    nv++ ;

    return p ;
}

. . .
char *gen_code (char *name)     // copia el argumento a un
{                               // string en memoria dinamica
    char *p ;
    int l ;

    l = strlen (name)+1 ;
    p = (char *) my_malloc (l) ;
    strcpy (p, name) ;

    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`).

Y también se incluye una sección de prototipos para otras funciones utilizadas y una variable global temporal (`temp`) para generar secuencias de traducción:

```
%{                          // SECCION 1 Declaraciones de C-Yacc

#include <stdio.h>
#include <ctype.h>            // declaraciones para tolower
#include <string.h>           // declaraciones para strings
#include <stdlib.h>           // declaraciones para exit ()

#define FF fflush(stdout);    // para forzar la impresion inmediata

int yylex () ;
void yyerror () ;
char *my_malloc (int) ;
char *gen_code (char *) ;

char temp [2048] ;
```

En cuanto al tipo de la pila del parser, como ya no necesitamos evaluar variables, ni tendrán un nombre de un solo carácter, por lo que el tipo index es inútil. Lo sustituimos por el atributo **`code`** de tipo char* (tipo string), que contendrá la secuencia de caracteres para el nombre de cada variable. Este atributo también se usará para almacenar y transmitir las traducciones en el proceso de análisis sintáctico. El atributo **`value`** se utilizará para almacenar valores numéricos. También se declaran varios token nuevos.

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 atributo `code de` tipo char* (tipo `string`), que contendrá la secuencia de caracteres del nombre de cada variable. El atributo `value` servirá para almacenar valores numéricos. También se declaran diversos *token*.

```
// Definitions for explicit attributes

typedef struct s_attr {
        int value ;
        char *code ;
        t_node *node ; // - for possible future use of AST
} t_attr ;

#define YYSTYPE t_attr

%}

// Definitions for explicit attributes

%token NUMBER
%token IDENTIF       // Identificador=variable
%token INTEGER       // identifica el tipo entero
%token STRING
%token MAIN          // identifica el comienzo del proc. main
%token WHILE         // identifica el bucle main
```

Esta definición sustituye la que permite usar atributos implícitos:

```
// Definitions for implicit attributes.
// USE THESE ONLY AT YOUR OWN RISK

/*

%union {                  // El tipo de la pila tiene caracter dual
    int value ;           // - valor numerico de un NUMERO
    char *code ;          // - para pasar los nombres de IDENTIFES
}

%token <value> NUMBER       // Todos los token tienen un tipo para la pila
%token <code> IDENTIF       // Identificador=variable
%token <code> INTEGER       // identifica la definicion de un entero
%token <code> STRING
%token <code> MAIN          // identifica el comienzo del proc. main
%token <code> WHILE         // identifica el bucle main

%type <...> Axiom ...

*/
```

Se han sustituido debido a:

- los atributos implícitos provocan algunos errores difíciles de depurar
- la estructura *struct* facilita trabajar con atributos explícitos simultáneos.

Especificaciones (Frontend)

Las especificaciones previas del fichero **trad1.y** proporcionado son:

- Hay una estructura jerárquica en la gramática que empieza con:
  - **Axioma**, se encarga de la recursividad (**r_axioma**) y deriva:
  - **Sentencia**, que a su vez deriva la asignaciones y las impresión de expresiones.
- La sentencia que contiene únicamente una expresión se ha eliminado. Aunque en el lenguaje C se permiten sentencias como **1+2;** no les sacaremos partido, y además pueden ser una fuente de conflictos.
- Podéis añadir vuestra solución para asignaciones encadenadas si disponéis de ella.
- El código de **trad1.y** utiliza código diferido. Dejaremos el posible uso de AST para más adelante.
- Estudiad las soluciones proporcionadas para:

```
a. termino ::= operando
b. termino ::= + operand
c. termino ::= - operand
d. operando ::= IDENTIF
e. operando ::= ( expresion )
```

Proponemos una secuencia de pasos para desarrollar esta práctica. Se recomienda abordar cada uno de los pasos de forma secuencial.

1. Incluid la definición de variables globales en la gramática.

   a. 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.

   b. Ampliad la gramática para contemplar la definición de una variable con inicialización 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.

   c. 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)** . El orden de impresión de las variables debe corresponder al de la declaración. Prestad atención a que se asignan valores constantes (numéricos), no expresiones evaluables. Esto es así porque para las variables globales en C no es posible evaluar dichas expresiones en tiempo de compilación.

   > Esto corresponde a la definición de variables globales. Recordamos que en C no está permitido inicializar variables con expresiones (con variables y funciones) en la instrucción en que es declarada puesto que el proceso se realiza en tiempo de compilación. La evaluación de expresiones se hace en tiempo de ejecución. Algunos compiladores sí disponen de una fase previa capaz de evaluar expresiones simples del tipo **int ab32 = 3+7 ;** y de sustituirlas por **int ab32=10;** pero sigue siendo una tarea realizada en tiempo de compilación.

[2] A partir de este punto comenzamos a traducir de C a Lisp

2. 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`

A través de la gramática es posible obligar a que exista una (única) función **`main`**.

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 en el *parser*. 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.

Aquí se indican varias cuestiones.

1) Hay que diseñar una gramática lo más jerárquica o estructurada posible. Los nombres de No Terminales deben escogerse con cuidado y deben ser representativos y significativos.

2) Esto evitará la aparición de errores típicos de diseños “enmarañados” o excesivamente complicados.

3) Se sugiere emplear una estructura de programa que empiece con la declaración de variables y luego con la definición de funciones.

4) La recomendación es definir las funciones en orden inverso de jerarquía, empezando con las funciones más sencillas y terminando por la principal, el **`main`**. Esto permitirá que más adelante el traductor tenga siempre referencia de las funciones que se usan porque ya se han definido previamente. Si se definen primero las funciones principales y luego las de inferior jerarquía, un compilador necesitará hacer averiguaciones sobre el tipo de las funciones que se llaman y que aún no se han definido, o emplear dos pasadas para realizar traducciones parciales y condicionadas.

El compilador de C requiere en estos casos de una declaración previa (prototipo) de las funciones. En Lisp deben definirse las funciones antes de ser usadas. Para evitarnos complicaciones con los prototipos usaremos programas en C con las funciones en orden jerárquico inverso.