Memoria de prácticas

Intérprete en Bison para lenguaje en pseudocódigo

Fernando Sánchez Delgado

Asignatura de Procesadores de Lenguajes
Grado en Ingeniría Informática
Especialidad en computación

Escuela Politécnica Superior
Universidad de Córdoba
Curso 2016-2017
20 Mayo 2017, Córdoba

Contents

1 Introducción

Este trabajo consiste en la realización de un intérprete para un lenguaje de pseudocódigo cuya especificación nos ha sido dada en el enunciado de la práctica. Asimismo se han añadido ciertas mejoras que se han visto oportunas para obtener tanto un lenguaje como un intérprete más robustos.

El intérprete admite dos modos de ejecución distintos, uno es el modo interactivo, el cual permite introducir código que será evaluado de inmediato. El otro modo consiste en proporcionar al intérprete un achivo del cual obtener el código a ejecutar.

En cuanto a la estructura de este documento, primero procederemos a hablar sobre el lenguaje, para después ir profundizando en los detalles de la implementación. Por último, se mostrarán algunos ejemplos y se expondrán las conclusiones obtenidas tras la realización de este trabajo.

2 Lenguage pseudocódigo

2.1 Componentes léxicos

2.1.1 Palabras reservadas

Las palabras reservadas del lenguaje no podrán ser utilizadas como identificadores, se han reservado las siguientes palabras:

```
_mod, _div, _o, _y, _no, leer, leer_cadena, escribir, escribir_cadena
si, entonces, si_no, fin_si, mientras, hacer, fin_mientras, repetir, hasta
para, desde, hasta, paso, fin_para, _borrar, _lugar
```

Además de esta palabras se reservadas, el intérpre añade algunas variables constantes, las cuales no puede ser modificadas:

```
PI, E, GAMMA, DEG, PHI
```

2.1.2 Identificador

Los identificadores pueden estar compuestos de letras, dígitos o guiones bajos. Sin embargo, se debe cumplir que: empiece por una letra, no acabe en guión bajo y no haya dos guiones bajos seguidos.

Ejemplo de identificador válido:

```
iden, iden_1a, ident_a_1_3
```

Ejemplo de identificadores no válidos:

```
_iden, iden_1_, iden__3
```

2.1.3 Número

Los números pueden ser de tipo entero, real o con notación científica.

Ejemplo números válidos:

```
38  5.289   3.1e10
```

Ejemplo números no válidos:

```
3.000.000   3,895   0x398
```

2.1.4 Cadena

Las cadenas se delimitan por comillas simples y pueden contener cualquier tipo de carácter a excepción de un salto de línea. Para permitir caracteres especiales se han incluido los siguientes código de escape:

```
\t      Tabulador
\n      Salto de línea
\'      Comilla simple
\\      Barra invertida
```

Además de las cadenas simples, también se ha añadido un nuevo tipo de cadena, la cadena multilínea. Esta cadena permite la inclusión de saltos de línea. Su sintaxis requiere de tres comillas simples delimitando cada extremo de la cadena.

Ejemplo de cadenas válidas:

```
'Cadena simple'
'\tCadena con caracteres escapados\n'
'''Cadena
multilínea
'''
```

2.1.5 Operador de asignación

El operador de asignación es `:=`, similar al usado en el lenguaje *Pascal*.

2.1.6 Operadores aritméticos

Los operadores aritméticos permitidos son:

- *Suma*: `+`, tanto unario como binario. Por ejemplo `+3` y `3+5`.
- *Resta*: `-`, tanto unaria como binaria. Por ejemplo `-7` y `2+6`.
- *Producto*: `*`, solo binario. Por ejemplo `3 * 7 = 21`.

- *División*: `/`, solo binaria. Por ejemplo `9 / 3 = 0`.
- *División entera*: `_div`, solo binaria. Por ejemplo `3 _div 2 = 1`.
- *Módulo*: `_mod`, solo binario. Por ejemplo `10 _mod 3 = 1`.
- *Potencia*: `**`, solo binaria. Por ejemplo `3 ** 2 = 9`.

2.1.7 Operador alfanumérico

El operador alfanumérico `||` se trata de un operador binario encargado de concatenar dos cadenas. Por ejemplo:

```
'hola ' || ' mundo' = 'hola mundo'
```

2.1.8 Operadores relacionales de números y cadenas

Los operadores relacionales mostrados a continuación admiten tanto números como cadenas. Existen los siguientes operadores:

- *Menor que*: `<`
- *Menor o igual que*: `<=`
- *Mayor que*: `>`
- *Igual que*: `=`
- *Distinto que*: `<>`

  Nota: cuando se usan para comparar cadenas se está realizando una comparación léxico gráfica. En concreto la implementada por la función `strcmp` de la librería estándar de C. *Referencia*

2.1.9 Operadores lógicos

Los operadores lógicos disponibles son:

- *Disyunción lógica*: `_o`
- *Conjunción lógica*: `_y`
- *Negación lógica*: `_no`

Ejemplos de uso de operadores lógicos:

```
(A >= 0) _y _no (control <> 'stop')
```

2.1.10 Comentarios

Existe dos tipos de comentarios, los multilínea (delimitados por `#`) y los de una línea (comienzan con `@`).

Ejemplos de comentarios:

```
@ Comentario de una línea

# Comentario
multilínea
#
```

2.1.11 Punto y coma

El punto y coma `;` se usa como carácter delimitador, para indicar el fin de una sentencia.

```
escribir_cadena('hola');
```

2.2 Sentencias

2.2.1 Asignación

La asignación se realiza desde una expresión númerica o alfanumérica hacia un identificador. Los identificadores pueden cambiar el tipo asociado que tengan previamente. No se puede asignar a las constantes declaradas por el intérprete.

Ejemplo de asignación:

```
cadena := 'hola';
numero := 38;
numero := cadena;  @ numero ahora vale 'hola'
```

2.2.2 Lectura

Para leer entrada por teclado por parte del usuario se pueden usar `leer` y `leer_cadena`. Para ambas se debe proporcionar un identificador para la variable en la que se deba guardar lo leído.

Solo si se está ejecutando el intérprete en modo interactivo, se mostrará al usuario una pista de lo que tiene que introducir. Por ejemplo `Cadena-->` o `Número-->`.

```
leer(dato);  @ el usuario introduce 5
(dato = 5) = true
```

2.2.3 Escritura

Para escribir por pantalla se puede usar `escribir` y `escribir_cadena`, pasando como parámetro la expresión cuya evaluación se mostrará por pantalla. Para hacer más robusto el intérprete y facilitar el uso, se permite que se usen de forma intercambiable, sin importar el tipo de la variable. La razón por la que se ha mantenido `escribir_cadena` es para evitar inconsistencias.

Solo si se está ejecutando en modo interactivo el intérprete, se mostrará el resultado precedido de `-->` para indicar que es la salida del comando.

```
escribir(3 + 5);  @ imprime 8
```

2.2.4 Sentencias de control

A continuación se muestran todas las sentencias destinadas a controlar el flujo de ejecución del programa.

2.2.4.1 Condicional

Se permite tanto con opción alternativa como sin ella.

si *condición*

entonces *sentencias*

si_no *sentencias*

fin_si

Ejemplo:

```
var := 3;
si (var = 3) entonces
    escribir('hola');
si_no
    escribir('adios');
fin_si;
@ escribe hola
```

2.2.4.2 Bucle *mientras*

Se ejecuta mientras se cumpla la condición. Evalúa la condición antes de realizar cada iteración.

mientras *condición* **hacer**

sentencias

fin_mientras

Ejemplo:

```
dato := 1;
mientras (dato < 5) hacer
    dato := dato +1;
fin_mientras;
```

2.2.4.3 Bucle *repetir*

Este bucle se ejecuta hasta que se cumpla la condición. Evalúa la condición después de realizar cada iteración.

repetir

sentencias

hasta *condición*

Ejemplo:

```
dato := 0;
repetir
    escribir('hola');
    dato := dato + 1;
hasta (dato = 3);
@ escribe hola 3 veces
```

2.2.4.4 Bucle *para*

Este bucle comienza asignando a una variable el valor inicial, e itera hasta que alcanza el valor final. En cada iteración aumenta la variable en la cantidad proporcionada como paso.

para *identificador*

desde *expresión numérica 1*

hasta *expresión numérica 2*

paso *expresión numérica 3*

hacer

sentencias

fin_para

Ejemplo:

```
para i
    desde 1
```

```
    hasta 3
    paso 1
    hacer
        escribir('hola');
fin_para;
@ escribe hola 3 veces
```

2.2.5 Comandos especiales

Se han incluido dos comando especiales para facilitar el desarrolo de programas. El comando `_borrar` borra el contenido de la pantalla actual. Por otro lado el comando `_lugar` permite colocar el cursor en la posición deseada de la pantalla.

```
_borrar;
_lugar(3,5);
@ borra la pantalla y sitúa el cursor en la posición (3,5)
```

3 Tabla de símbolos

La tabla de símbolos usada para esta implementación es la que se proporcionaba. Es decir, se ha implementado la tabla de símbolos como una lista simplemente enlazada. Se ha estimado que el rendimiento y simplicidad proporcionados por esta implementación son suficientes para el uso apropiado del intérprete.

A continuación se muestra la estructura usada para representar cada nodo de la lista:

```
typedef struct Symbol { /* entrada en la tabla de simbolos */
                    char *nombre;
                    short tipo;  /* VAR, FUNCION, INDEFINIDA */
                    short subtipo; /* STRING, NUMBER */
                    struct {
                       double val;        /* si es NUMBER */
                       char *str;         /* si es STRING */
                       double (*ptr)();   /* si es FUNCION */
                     } u;
                        struct Symbol * siguiente;
                   } Symbol;
```

4 Análisis léxico

El fichero proporcionado a *Flex* para el análisis léxico es `lexico.l`. A continuación se procede a realizar una análisis de las partes que componen al fichero.

4.1 Definiciones

En esta parte se definen tanto las expresiones regulares como los estados que se van a usar. Se definen varias expresiones regulares que pueden resultar útiles. Además también se define el estado `COMENTARIO` usado para poder procesar los comentarios multilínea.

```
12 /* definiciones regulares */
13 numero        [0-9]
14 letra         [a-zA-Z]
15 identificador {letra}(({letra}|{numero}|_({letra}|{numero}))*)?
16 identificador_malo ({letra}|{numero}|_)+
17 espacio         [\t\n]
18
19 %x COMENTARIO
```

4.2 Reglas

A partir de la línea 20 del fichero se encuentran todas las reglas usadas por flex para realizar el análisis. En la primera parte se definen las reglas para leer comentarios, cadenas y números. Mientras que en la segunda parte se encuentran las reglas para reconocer operadores y los comandos especiales.

> *Nota:* Se ha omitido el código ejecutado por las reglas para reducir la longitud del fragmento.

```
[ \t] { ; } /* saltar los espacios y los tabuladores */

"#" {BEGIN COMENTARIO;}
<COMENTARIO>"#" {BEGIN 0;} /* Vuelve al estado normal */
<COMENTARIO>\n   {lineno++;}
<COMENTARIO>.   {;}

"@".*\n {;} /* En los comentarios se ignoran todos los caracteres */

'([^'\n]|\\')*' {...}

'''([^']|\\')*''' {...}

{numero}+\.?|{numero}*\.{numero}+|{numero}*(\.{numero}*)?[eE]{numero}+ {...}

{identificador} {...}

"||"  {return CONCATENAR;}
">="    {return MAYOR_IGUAL;}
"<="    {return MENOR_IGUAL;}
```

```
"=" {return IGUAL;}
"<>"    {return DISTINTO;}
 ...
```

5 Análisis sintáctico

5.1 Estructura del fichero

El fichero utilizado por *Bison* para definir la gramática es `ipe.y`. A continuación se va a proceder a explicar las partes. Se ha evitado incluir código ya que se haría inncesariamente extenso el documento.

- En la primera parte del fichero se pueden incluir declaraciones en C, en este caso son algunos *include* y *define*.
- La segunda parte (líneas 26-42) de un fichero de *Bison* son las declaraciones, en este parte se definen los token que se usarán así como las preferencias.
- La tercera parte (líneas 45-147) contiene todas las reglas de la gramática que debe tener en cuenta *Bison*.
- La cuarta parte del fichero está destinada a código en C que se desee incluir.

5.2 Modificaciones realizadas

Se han añadido a la parte de declaraciones los nuevos símbolos terminales y no terminales necesarios.

```
%token <sym> NUMBER STRING VAR CONSTANTE FUNCION0_PREDEFINIDA FUNCION1_PREDEFINIDA FUNCI
%token <sym> PRINT WHILE DO WHILE_END IF THEN ELSE IF_END READ REPEAT UNTIL FOR FROM STE
%token <sym> READ_STR PRINT_STR CLEAN POSITION
%type <inst> stmt asgn expr stmtlist cond mientras si repetir para variable end
%right ASIGNACION
%left O_LOGICO
%left Y_LOGICO
%left MAYOR_QUE MENOR_QUE MENOR_IGUAL MAYOR_IGUAL DISTINTO IGUAL
%left '+' '-' CONCATENAR
%left '*' '/' DIVIDIR_INT MODULO
%left UNARIO NEGACION
%right POTENCIA
```

Se han añadio las siguientes reglas de producción:

```
| READ_STR '(' VAR ')' {code2(leercadena,(Inst)$3);}
| CLEAN         {code(borrarpantalla);}
```

```
| POSITION '(' NUMBER ',' NUMBER ')' {code3(posicion,(Inst)$3,(Inst)$5);}
...
| repetir stmtlist end UNTIL cond end
...
| para variable FROM expr end UNTIL expr end STEP expr end DO stmtlist end FOR_END
...
variable: VAR {code((Inst)$1); $$=progp;}
        ;
...
repetir:     REPEAT     {$$= code3(repeatcode,STOP,STOP);}
        ;

para:        FOR        {$$= code3(forcode,STOP,STOP); code2(STOP,STOP);}
...
```

Además se han modificado algunas reglas de producción ya existentes. Entre ellas se ha excluido la que permitía bloques delimitados por llaves. También se han modificado las estructuras de control para incluir *token* que indiquen el fin, por ejemplo `fin_si` para el condicional.

En la función `main` se han realizado algunas modificiones para permitir que exista una variable global que indique al resto de funciones si se está ejecutando el intérprete en modo interactivo o *batch*.

Bucle repetir

Para el *bucle repetir*, sólo es necesario guardar la dirección de la condición de parada del bucle y la dirección de la siguiente instrucción. Para acceder al cuerpo del bucle, se accede a la posición siguiente a donde se guarda la dirección de la siguiente instrucción.

```
| repetir stmtlist end UNTIL cond end
          {
            ($1)[1]=(Inst)$5; /* condición de repetir */
            ($1)[2]=(Inst)$6; /* siguiente instruccion */
          }
...
repetir:     REPEAT     {$$= code3(repeatcode,STOP,STOP);}
```

Estructuras de las intrucciones:

```
     STOP <1>->
     STOP <2>->
     cuerpo
<1>->cond
     STOP
<2>->...
```

Bucle para

En este bucle, lo que se necesita guardar en las posiciones contiguas al código del bucle, es el valor de parada, el paso, el cuerpo del bucle y el de la siguiente instrucción. Se supone que a continuación estarán en memoria la dirección de la variable usada para el bucle y la de la expresión que inicializa la variable.

```
| para variable FROM expr end UNTIL expr end STEP expr end DO stmtlist end FOR_END
          {
            ($1)[1]=(Inst)$7;  /* valor final */
            ($1)[2]=(Inst)$10; /* paso entre cada iteracion */
            ($1)[3]=(Inst)$13; /* cuerpo del bucle */
            ($1)[4]=(Inst)$14; /* end */
          }
...
para:        FOR        {$$= code3(forcode,STOP,STOP); code2(STOP,STOP);}
```

Estructuras de las intrucciones:

```
     STOP <1>->
     STOP <2>->
     STOP <3>->
     STOP <4>->
     VAR
     expr1
     STOP
<1>->expr2
     STOP
<2>->expr3
     STOP
<3>->cuerpo
     STOP
<4>->...
```

6 Funciones auxiliares

En esta sección se describen las funciones auxiliares que se han tenido que crear para ofrecer la funcionalidad que actualmente tiene el intérprete. Todas estas funciones se encuentran en el fichero `code.c`.

- `execute`(l.83): se ha añadido control de errores para evitar errores en programas con una mala sintaxis.
- `dividir_int`(l.151): función encargada de hacer la división entera.
- `escribir`(l.168): modificada para poder imprimir cadenas.
- `borrarpantalla`(l.208): función asociada al comando `_borrar`.

- `posicion`(l.213): función asociada al comando `_lugar`.
- `concatenar`(l.342): función asociada al operador `||`.
- `leercadena`(l.399): función asociada al comando `leer_cadena`.
- `evaluarcadena`(l.427): función encargar de sustituir los caracteres escapados en una cadena.
- `mayor_que`(l.456): adaptada para aceptar cadenas.
- `menor_que`(l.485): adaptada para aceptar cadenas.
- `igual`(l.514): adaptada para aceptar cadenas.
- `mayor_igual`(l.542): adaptada para aceptar cadenas.
- `menor_igual`(l.571): adaptada para aceptar cadenas.
- `distinto`(l.599): adaptada para aceptar cadenas.
- `repeatcode`(l.726): código del bucle *repetir*.
- `forcode`(l.747): código del bucle *para*.

En el fichero `symbol.c` en la línea 47 comienza la función `install`, la cual se ha modificado para poder instalar cadenas.

7 Modo de obtención del intérprete

Para poder compilar con éxito el intérprete son necesarios los siguientes ficheros:

- `code.c`: incluye el código para todas las funciones que ejecutan funcionalidad del pseudocódigo.
- `init.c`: inicializa las constantes, palabras clave y las funciones de disponibles en el pseudocódigo.
- `ipe.h`: cabecera para todo el programa, define las estructuras usadas.
- `ipe.y`: fichero de *Bison* con las reglas de la gramática.
- `lexico.l`: fichero de *Flex* con las reglas del léxico.
- `math.c`: fichero que define las funciones matemáticas disponibles.
- `symbol.c`: fichero con funciones que manipulan la tabla de símbolos.

El `Makefile` usado se ha creado a partir de algunas modificaciones sobre el fichero proporcionado al principio. Los cambios necesarios en cuestión, han sido:

- Cambiar la variable `FUENTE` a `ipe`, para cambiar el nombre de los ficheros principales.
- Borar el número de versión en `VERSION` para que no necesiten sufijo numérico los ficheros.

El proceso de compilación coniste en generar el archivo de *Bison* y compilarlos, generar el archivo de *Flex* y compilarlo. Luego se compilan el resto de archivos `.c` y finalmente se unen todos los archivos objeto para crear el ejecutable del intérprete.

8 Modo de ejecución del intérprete

El intérprete dispone de dos modos de ejecución. Se ha añadido funcionalidad para que el comportamiento sea ligeramente distinto en cada modo, de esta forma se adapta mejor a las necesidades de cada modo.

8.1 Modo interactivo

En este modo, el intérprete se encarga de evaluar cada una de las órdenes introducidas por el usuario de forma inmediata. En este modo las salidas por pantalla usandos las funciones de escritura, se prefijan con `--->`. Y las llamadas a `leervariable` y `leercadena` piden los datos con los prefijos `Número-->` y `Cadena-->` respectivamente.

Ejemplo de salida del modo interactivo:

```
dato:=3;
escribir(dato);
     ---> 3
escribir_cadena('hola ' || 'mundo');
     ---> hola mundo
```

8.2 Modo *batch*

En este modo el intérprete toma un fichero con el pseudocódigo y lo ejecuta. Cuado este modo está activo, se quitan las pistas visuales para el usuario en las funciones de escritura y lectura.

Ejemplo llamada a modo batch:

```
$ ./ipe.exe fichero_fuente.e
```

9 Ejemplos

9.1 Ejemplos previos

A continuación se listan los ejemplos proporcionados por el profesor:

- `ejemplo_1_saluda.e`: pide el nombre al usuario y lo saluda con un mensaje.

- `ejemplo_2_factorial.e`: pide un número al usuario y calcula el factorial.
- `ejemplo_3_mcd.e`: pide dos números al usuario y calcula su máximo común divisor.
- `ejemplo_4_menu_inicial.e`: muestra un menú de prueba al usuario.
- `ejemplo_5_menu_completo.e`: muestra un menú que permite obtener el factorial y el máximo común divisor.
- `ejemplo_6_Intercambiar_tipos.e`: fichero cuyo objetivo es comprobar si las variables tienen carácter dinámico.

9.2 Ejemplos añadidos

Se han creado tres nuevos ejemplos, siendo el últimos ellos `ejemplo_9_fib-prim.e` un agregado de los anteriores junto con un menú.

9.2.1 Ejemplo 1

En este ejemplo se pide un número al usuario, y se calculan tantos términos de la serie según la magnitud del número leído. El código se encuentra en el fichero `ejemplo_7_fibonacci.e`.

```
_borrar;
_lugar(10,10);
escribir_cadena('Serie de fibonacci\n');
_lugar(11,10);
escribir_cadena('Introduce el número de términos--> ');
leer(N);

a := 0;
b := 1;

_borrar;
escribir_cadena('Primeros ');
escribir(N);
escribir_cadena(' términos de la serie:\n');

para i desde 1 hasta N paso 1
hacer
  aux := a;
  a := b;
  b := aux + b;
  escribir_cadena('\t');
  escribir(aux);
  escribir_cadena('\n');
fin_para;
```

9.2.2 Ejemplo 2

En este ejemplo se pide al usuario un número, y se comprueba si es primo o no. El código se encuentra en el fichero `ejemplo_8_primalidad.e`.

```
_borrar;
_lugar(10,10);
escribir_cadena('Test de primalidad\n');
escribir_cadena('Introduce un número--> ');
leer(N);

primo := 1;

para i desde 2 hasta N-1 paso 1
hacer
  si ((N _mod i) = 0) entonces
    primo := 0;
  fin_si;
fin_para;

_borrar;
_lugar(10,10);
si (primo = 1) entonces
  escribir(N);
  escribir_cadena(' es primo\n');
si_no
  escribir(N);
  escribir_cadena(' no es primo\n');
fin_si;
```

10 Conclusiones

10.1 Conclusión sobre el trabajo

Realizar este trabajo ha servido tanto para afianzar los contenidos vistos en las clases de teoría, como para pero ver alguna de las aplicaciones prácticas de los contenidos vistos.

También se comprende algo más el funcionamiento de los intérpretes de lenguajes y compiladores. Aunque no se llega a profundizar demasiado, sirve para respetar y admirar la gran complejidad de las herramientas que proporcionan los lenguajes de programación que usamos a diario.

Por otro lado, este trabajo también nos ha permitido afianzar las habilidades prácticas para implementar programas que analicen datos estructurados. Por ejemplo ficheros de

configuración o formatos para transmisión de datos (ej.json). O sentar las bases sobre las que seguir aprendiendo para poder participar en el desarrollo de lenguajes de programación actuales o futuros.

10.2 Puntos débiles y fuertes del intérprete

Los principales puntos fuertes de este intérprete son su capacidad para leer ficheros e interactuar con el usuario. Además el hecho de que acepte un pseudocódigo en español puede facilitar el entendimiento a algunas personas que tienen dificultad con el inglés.

También se puede resaltar el hecho de que al ser un intérprete bastante simple, es adecuado para labores educativas. Ya que permite pronfundizar en los distintos aspectos de los intérpretes sin llegar a aumentar la complejidad de forma excesiva.

Como puntos débiles se pueden destacar su número limitado de tipos, así como una incapacidad de convertir entre ellos. Además, la incapacidad de definir funciones propias limita seriamente la capacidad del usuario para crear programas complejos y que cumplan los requisitos de reutilización y claridad en la estructua del código.

Los otros puntos débiles del intérprete se pueden encontrar en la limitación en la longitud de los programas, y el la mala gestión de la memoria que realiza.