UNIVERSIDAD D CÓRDOBA

Intérprete de Pseudocódigo en Español (IPE)

Universidad de Córdoba, Escuela Politécnica Superior de Córdoba.
Grado en Ingeniería Informática, especialidad Computación.
Procesadores de Lenguajes.
Tercer curso, segundo cuatrimestre, curso 2013/2014.
Junio 2014.

Daniel Luque Quintana
David Luque Quintana

Índice

Introducción

El trabajo al que se corresponde esta memoria es un intérprete de pseudocódigo. Se ha programado tanto el análisis léxico usando archivos de Flex (.l) y los archivos correspondientes a la parte sintáctica, en este caso usando Bison (.y).

El intérprete es capaz de reconocer ciertos comandos que deben estar en español e introducidos por el programador harán que pueda implementar código simplemente sabiendo las nociones de pseudocódigo.

En el siguiente documento, se tratarán las palabras usadas para crear el lenguaje de pseudocódigo, la tabla de símbolos, el análisis léxico y sintáctico, las funciones auxiliares, modo de obtención y ejecución del intérprete, ejemplos y conclusiones.

Lenguaje de pseudocódigo

A continuación se va a desarrollar los distintos componentes léxicos y semánticos del lenguaje en pseudocódigo utilizado en el intérprete.

Componentes léxicos o tokens

Palabras reservadas

Las palabras reservadas son aquellas que no pueden ser usadas como identificadores, ya que están reservadas para otros usos. No se distingue entre mayúscula y minúscula. La lista de palabras reservadas es la siguiente: #mod, #o, #y, #no, leer, leer_cadena, escribir, escribir_cadena, si, entonces, si_no, fin_si, mientras, hacer, fin_mientras, repetir, hasta, para, desde, paso, fin_para, borrar y lugar.

Identificadores

Al igual que las palabras reservadas, no habrá distinción entre mayúscula y minúscula. Estarán compuestos por una serie de letras, dígitos y el subrayado. Deben comenzar por una letra y no podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.

- Ejemplo válido: dato, dato_1, dato_1_a.
- Ejemplo inválido: _dato, dato_, dato__1.

Número

El intérprete aceptará números enteros, reales de punto fijo y reales con notación científica. Además serán tratados todos conjuntamente como números.

- Enteros: 5, 20.
- Reales: 0.5, 0.012.
- Notación científica: 1e5 ($1 \cdot 10^5$)

Cadena

Las cadenas estarán compuestas por una serie de caracteres delimitados por comillas simples y deberá permitir la inclusión de comilla simple con la utilización de la barra (\): *'Ejemplo de cadena con \' comillas \' simples'.*

Operadores aritméticos

Los operadores aritméticos que acepta el intérprete son, ordenados según su prioridad: suma, resta, producto, división, módulo y potencia.

- Suma (+): se permiten tanto números unarios (+2) como binarios (1+2).
- Resta (-): al igual que la suma, permite números unarios (-1) como binarios (2-3).
- Producto (*): 3*3 = 9
- División(/): 3/3 = 1
- Módulo (#mod): 10 #mod 3 = 1
- Potencia (**): 2**3 = 8

Operadores alfanuméricos

El único operador alfanumérico implementado es el de la concatenación. Permite concatenar dos cadenas alfanuméricas en una.
Concatenación (||): cadena = 'Hola'.

Operadores relacionales de números y cadenas

Se han implementado operadores relacionales tanto para números como para cadenas:

- Menor que (<).
- Menor o igual que (<=).
- Mayor que (>).
- Mayor o igual que (>=).
- Igual que (==).
- Distinto que (<>).

Operadores lógicos

Se han implementado los operadores lógicos básicos:

- Disyunción lógica (#o).
- Conjunción lógica (#y).
- Negación lógica (#no).

Comentarios

Se han implementado dos formas de añadir comentarios al pseudocódigo: de una línea o de varias líneas. Todo lo que siga al carácter @ hasta el final de la línea será un comentario simple o de una línea, mientras que para hacer un comentario de varias líneas será necesario delimitarlo con llaves { }

Punto y coma

El carácter punto y coma *;* se utilizará para indicar el fin de una sentencia.

Sentencias

Asignación

Se utiliza la sentencia *identificador = expresión*. Declara a identificador como variable y le da el valor de la expresión. Esta variable puede ser numérica o alfanumérica, dependiendo de la expresión que se le asigne.

- Las expresiones numéricas se forman con números, variables numéricas y operadores numéricos.
- Las expresiones alfanuméricas se forman con cadenas, variables alfanuméricas y el operador alfanumérico de concatenación (||).

Lectura

Nos permite introducir datos desde el teclado y asignárselos a una variable. Existen dos tipos de lectura:

- Leer(identificador): se declara identificador como una variable númerica y le asigna el número leído.
- Leer_cadena(identificador): se declara identificador como una variable alfanumérica y se le asigna la cadena leída, sin comillas.

Escritura

Mediante esta sentencia podemos imprimir datos por pantalla. Al igual que en la lectura, existen dos tipos de escritura:

- Escribir(identificador): escribe por pantalla el valor de la expresión numérica del identificador.
- Escribir_cadena(identificador): escribe por pantalla la cadena, sin comillas exteriores, de la expresión alfanumérica del identificador. Se debe permitir la interpretación de saltos de línea (\n) y tabuladores (\t) que pueden aparecer en la expresión alfanumérica.

Sentencias de control

- Sentencia condicional simple
  si condición
  entonces sentencias
  fin_si

- Sentencia condicional compuesta
  si condición
  entonces sentencias
  si_no sentencias
  fin_si

- Bucle “mientras”
  mientras condición **hacer**
  sentencias
  fin_mientras

- Bucle “repetir”
  repetir
  sentencias
  hasta condición

- Bucle “para”
  para identificador
  desde expresión numérica 1
  hasta expresión numérica 2
  paso expresión numérica 3
  hacer
  sentencias
  fin_para

Comandos especiales

- Borrar: borra la pantalla.
- Lugar(expresión numérica 1, expresión numérica 2): coloca el cursor de la pantalla en las coordenadas indicadas por las expresiones numéricas.

Tabla de símbolos.

Definida en *ipe.h* se trata de una lista enlazada con una serie de campos:

- Nombre: nombre de la entrada, tipo char *.
- Tipo: el tipo de la entrada. Las opciones son NUMBER, VAR, FUNCION, INDEFINIDA o CONSTANTE. Campo de tipo short.
- Estructura u:
    - Valor: valor de la entrada, si es de tipo numérica. Tipo double.
    - Puntero a función de tipo double.
    - Cadena: si es una variable alfanumérica. Tipo char *

Análisis léxico

Las expresiones regulares definidas para el análisis léxico se encuentran en el fichero de Flex *lexico.l* y son las siguientes:

- digito [0-9]
- numero {digito}+(\.{digito}+)?(E[+\-]?{digito}+)?
- letra [a-zA-Z]
- subrayado [_]
- identificador {letra}({letra}|{numero}|{subrayado}({letra}|{numero}))*
- cadena """([^']|\\\')*"""
- espacio [\t\n]
- comentario [\@].*{espacio}*

Análisis sintáctico

Definido en el fichero de Bison *ipe.y* consta de símbolos terminales y no terminales, reglas de producción y acciones semánticas.

Símbolos terminales

- NUMBER: hace referencia a un número entero, real o en notación científica.
- VAR: hace referencia a una variable donde almacenar datos.

- CADENA: símbolo que guarda una cadena alfanumérica.
- CONSTANTE: símbolo que guarda un valor constante.
- FUNCION0_PREDEFINIDA: hace referencia a una función que no recibe ningún parámetro.
- FUNCION1_PREDEFINIDA: hace referencia a una función que recibe un parámetro.
- FUNCION2_PREDEFINIDA: hace referencia a una función que recibe dos parámetros.
- INDEFINIDA: hace referencia a aquellos elementos que han sido guardados en la pila pero sin un tipo.
- READ: hace referencia a la función *leer*.
- READ_CHAR: hace referencia a la función *leer_cadena*
- PRINT: hace referencia a la función *escribir.*
- PRINT_CHAR: hace referencia a la función *escribir_cadena*
- IF: hace referencia a la función *si*
- THEN, ELSE, END_IF: utilizados en la función *si*
- WHILE: hace referencia a la función *mientras*
- DO: utilizado en la función *mientras* y *para*
- END_WHILE: utilizado en la función *mientras*
- REPEAT, UNTIL: utilizado en la función *repetir*
- FOR: hace referencia a la función *para*
- SINCE, INC, END_FOR: utilizados en la función *para*
- TOKEN_LUGAR: hace referencia a la función *lugar.*
- TOKEN_BORRAR: hace referencia a la función *borrar*
- ASIGNACION: hace referencia a la función de asignación *asgn*
- O_LOGICO: hace referencia a la función de disyunción.
- Y_LOGICO: hace referencia a la función de conjunción.
- MAYOR_QUE MENOR_QUE MENOR_IGUAL MAYOR_IGUAL IGUAL DISTINTO_QUE
- CONCATENACION: hacen referencia a las funciones de operadores relacionales.
- '+' '-' '*' '/' MODULO POTENCIA UNARIO NEGACION '^': hacen referencia a las funciones de operadores matemáticos.

Símbolos no terminales

- stmt: declaración de una sentencia.
    - Ejemplo: | *PRINT '(' expr ')' {code(escribir); $$ = $3;}*
- asgn: asigna una variable o constante a una expresión.
    - Ejemplo: *VAR ASIGNACION expr { $$=$3; code3(varpush,(Inst)$1,assign);}*
- expr: realiza las llamadas a las distintas funciones.
    - Ejemplo: | *expr '+' expr {code(sumar);}*
- stmtlist: crea una lista de declaraciones.
    - Ejemplo: | *stmtlist stmt FIN_SENTENCIA*
- cond: usada para crear condiciones usadas en las estructuras de control.
    - Ejemplo: *'(' expr ')' {code(STOP); $$ =$2;}*
- mientras, repetir, si, para: genera las llamadas a las funciones con el mismo nombre.
    - Ejemplo: *mientras: WHILE {$$= code3(whilecode,STOP,STOP);}*

- fin: crea un símbolo epsilon.
    - Ejemplo: *fin : {code(STOP); $$ = progp;}*
- variable: se encarga de instalar una variable. Usada en el bucle *para.*
    - Ejemplo: *variable: VAR {code((Inst)$1); $$=progp;}*

Reglas de producción

```
list :
      | list stmt  FIN_SENTENCIA
      | list error FIN_SENTENCIA
      ;

stmt :
      | asgn
      | PRINT '(' expr ')'
      | READ '(' VAR ')'
      | READ_CHAR '(' VAR ')'
      | PRINT_CHAR '(' expr ')'
      | TOKEN_LUGAR '(' expr ',' expr ')'
      | TOKEN_BORRAR
      | mientras cond DO stmtlist END_WHILE fin
      | si cond THEN stmtlist END_IF fin
      | si cond THEN stmtlist fin ELSE stmtlist END_IF fin
      | para variable SINCE expr fin UNTIL expr fin INC expr fin DO stmtlist END_FOR fin
      | repetir stmtlist fin UNTIL cond fin
      ;

asgn :    VAR ASIGNACION expr
        | CONSTANTE ASIGNACION expr
          ;

variable:  VAR
      ;

cond :    '(' expr ')'
      ;

mientras:    WHILE
      ;

repetir:    REPEAT
      ;
```

```
si:       IF
        ;

fin :
        ;

para:   FOR
        ;

stmtlist:
        | stmtlist stmt FIN_SENTENCIA
        ;

expr :    NUMBER
        | VAR
        | CONSTANTE
        | asgn
        | FUNCION0_PREDEFINIDA '(' ')'
        | FUNCION1_PREDEFINIDA '(' expr ')'
        | FUNCION2_PREDEFINIDA '(' expr ',' expr ')'
        | '(' expr ')'
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr MODULO expr
        | expr POTENCIA expr
        |'-' expr %prec UNARIO
        |'+' expr %prec UNARIO
        | expr MAYOR_QUE expr
        | expr MAYOR_IGUAL expr
        | expr MENOR_QUE expr
        | expr MENOR_IGUAL expr
        | expr IGUAL expr
        | expr DISTINTO_QUE expr
        | expr Y_LOGICO expr
        | expr O_LOGICO expr
        | NEGACION expr
        | CADENA
        | expr CONCATENACION expr
           ;
```

Acciones semánticas

A continuación se van a detallar las estructuras de control. En las sentencias, el orden de las palabras viene indicado mediante el símbolo $.

Estructura de control if

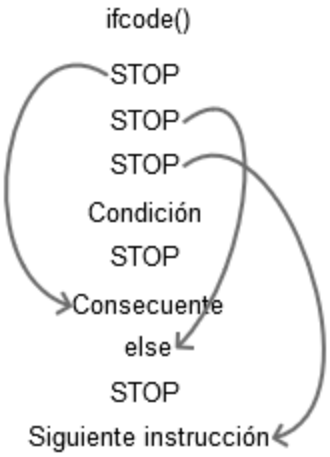


Figura 1. Estructura de control if

si cond THEN stmtlist fin ELSE stmtlist END_IF fin
{
	($1)[1]=(Inst)$4; Cuerpo del if
	($1)[2]=(Inst)$7; Cuerpo del else
	($1)[3]=(Inst)$9; Siguiente instrucción del if-else
}

Estructura de control while

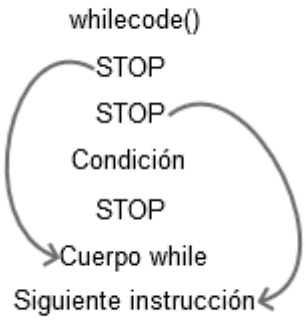


Figura 2. Estructura de control while

```
mientras cond DO stmtlist END_WHILE fin
{
        ($1)[1]=(Inst)$4; Cuerpo del while
        ($1)[2]=(Inst)$6; Siguiente instrucción del while
}
```

Estructura de control for

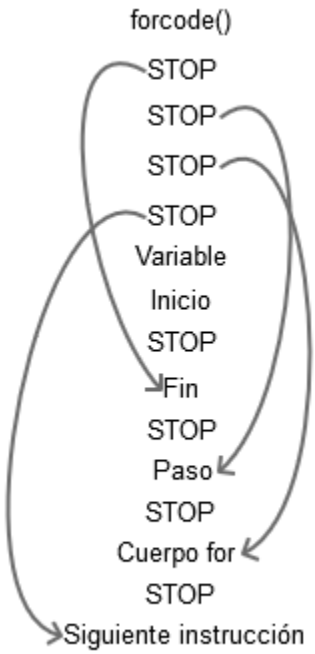


Figura 3. Estructura de control for

para variable SINCE expr fin UNTIL expr fin INC expr fin DO stmtlist END_FOR fin
{
	($1)[1] = (Inst)$7; Condición de parada
	($1)[2] = (Inst)$10; Incremento
	($1)[3] = (Inst)$13; Cuerpo del for
	($1)[4] = (Inst)$15; Siguiente instrucción del for
}

Estructura de control repetir

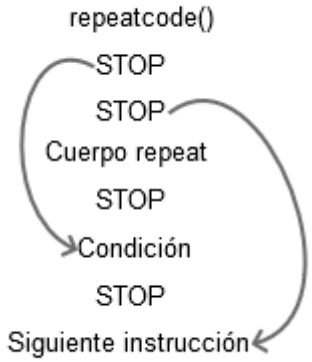


Figura 4. Estructura de control repeat

```
repetir stmtlist fin UNTIL cond fin
{
        ($1)[1]=(Inst)$5; Condición del bucle
        ($1)[2]=(Inst)$6; Siguiente instrucción del repeat
}
```

Funciones auxiliares

- Log(x): calcula el logaritmo neperiano de un número real.
- Log10(x): calcula el logaritmo decimal de un número real.
- Exp(x): calcula el exponencial de un número real.
- Sqrt(x): calcula la raíz cuadrada de un número real.
- Integer(x): calcula la parte entera de un número real.
- Azar(): calcula un número aleatorio.
- Atan2(x,y): calcula el arcotangente doble.
- errcheck(d,s): comprueba si se ha producido un error al realizar la evaluación matemática del primer parámetro. Si no hay error, entonces devuelve el primer parámetro; en caso contrario muestra el error producido.
- assign(): asigna el valor superior de la pila al siguiente valor.
- constpush(): mete una constante numérica en la pila.
- charpush(): mete una cadena alfanumérica en la pila.
- dividir(): divide los dos valores superiores de la pila y guarda su resultado en la pila.
- escribir(): saca de la pila el valor superior y lo escribe por pantalla.
- escribir_cadena(): igual que la función escribir, pero escribe por pantalla una cadena alfanumérica.

- eval(): evalúa la variable de la parte superior de la pila, indicando si es una cadena o un número y escribiendo su valor.
- funcion0(): evalúa una función predefinida sin parámetros.
- funcion1(): evalúa una función predefinida con un parámetro.
- funcion2(): evalúa una función predefinida con dos parámetros.
- modulo(): calcula el resto de la división entera del segundo valor de la pila por el de la cima, y guarda el resultado en la cima de la pila.
- multiplicar(): calcula el producto de los dos valores superiores de la pila, y guarda el resultado en la cima.
- negativo(): niega el valor superior de la pila.
- positivo(): devuelve el valor positivo del valor superior de la pila.
- potencia(): calcula la potencia del segundo valor de la pila, elevado a la cima de la pila, y guarda el resultado en la cima.
- restar(): calcula la diferencia entre el segundo valor de la pila y la cima, y guarda el resultado en la cima.
- sumar(): calcula la suma de los dos valores superiores de la pila y guarda el resultado en la cima.
- concatenar(): concatena dos cadenas en una y guarda el resultado en la pila.
- varpush(): mete una variable en la pila.
- leervariable(): lee una variable introducida por teclado y la guarda en la pila.
- leer_cadena(): lee una cadena alfanumérica por teclado y la guarda en la pila.
- borrarpantalla(): limpia la pantalla.
- colocarCursor(): color el cursor en una posición determinada.

Modo de obtención del intérprete

Descripción de cada fichero

- ipe.h: fichero donde se define la tabla de símbolos, la pila y los prototipos de las funciones.
- ipe.y: fichero Bison que genera el analizador sintáctico.
- code.c: fichero que contiene las funciones necesarios para el funcionamiento del intérprete. Son las acciones del código intermedio.
- init.c: fichero que contiene las palabras reservadas, constantes y la función init() que inserta en la tabla de símbolos.
- lexico.l: fichero de Flex que genera el analizador léxico.
- macros.h: fichero que incluye macros relacionadas con la impresión por pantalla.
- math.c: fichero que contiene funciones matemáticas.
- symbol.c: fichero donde se definen las funciones relacionadas con la tabla de símbolos.
- makefile: fichero utilizado para crear el intérprete.
- auxiliar.c: este fichero contiene una función que convierte las letras mayúsculas en minúsculas.

Descripción de makefile

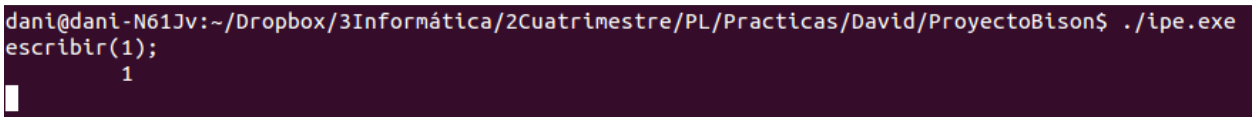
El fichero makefile es un fichero que permite compilar diversos ficheros de código escribiendo únicamente *make* en el terminal. En el makefile facilitado para el trabajo se encuentran llamadas a gcc, bison y flex (compilador, analizador sintáctico y léxico respectivamente).
Primero se compila con bison el analizador sintáctico, que posteriormente se usará con flex para crear el analizador léxico. Por último se usa el compilador gcc para crear los objetos y, en último paso, el ejecutable ipe.exe

Se incluye además una función clean, para eliminar todos los archivos objectos creados (.o) simplemente escribiendo en el terminal *make clean*.

Modo de ejecución

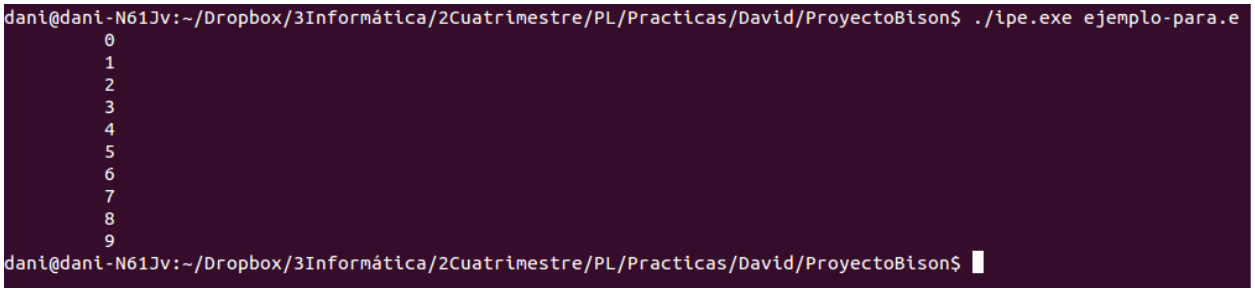
Interactiva

Mediante este modo se escriben las instrucciones por terminal, previa ejecución del ejecutable ipe.



A partir de un fichero

Podemos ejecutar las instrucciones desde un fichero. Para ello ejecutamos el ejecutable ipe y escribimos a continuación el nombre del fichero que contiene las instrucciones. Estos ficheros deben de tener la extensión .e.



Ejemplos

Archivo: ejemplo-if1.e

```
Escribe un número:
Valor--> 2
El numero es par
```

Archivo: ejemplo-if2.e

```
Escribe un número:
Valor--> 2
El número es positivo.
```

Archivo: ejemplo-mientras.e

```
Introduce un numero positivo
Valor--> 10
10
Introduce otro número positivo. Si quieres parar introduce un numero negativo
Valor--> 20
20
Introduce otro número positivo. Si quieres parar introduce un numero negativo
Valor--> -1
Adios
```

Archivo: ejemplo-para.e

```
Cuenta atrás para el lanzamiento. Introduce tiempo:
Valor--> 10
Segundos restantes:
10
Segundos restantes:
9
Segundos restantes:
8
Segundos restantes:
7
Segundos restantes:
6
Segundos restantes:
5
Segundos restantes:
4
Segundos restantes:
3
Segundos restantes:
2
Segundos restantes:
1
Segundos restantes:
0
Lanzamiento!
```

Archivo. ejemplo-repetir.e

```
Numeros pares menores de 50
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
```

Conclusiones

La conclusión más clara que tenemos al realizar el trabajo es que hemos aumentado nuestros conocimientos sobre cómo puede funcionar un lenguaje de interpretación de este estilo, de lo complejo que llega a ser en algunos casos y del cuidado que hay que tener para que algo que en una zona de código funciona bien es capaz de parar la ejecución en otra parte. Esto es por ejemplo, al unir en una misma variable el valor numérico con el de cadena, al declarar el tamaño de la cadena y asignarlo, cuando sólo introducimos el valor numérico teníamos problemas de violación de segmento que conseguimos arreglar.

El punto débil del código es la corrección de errores ya que se ha quedado para el final y no hemos tenido tiempo material para hacerlo correctamente, pero hemos intentado que al menos el resto del código sea robusto y no tenga fallos para intentar al menos minimizar los errores.