Tema 10 | Intérpretes

Contenidos

Bibliografía básica

9-Feb-2010

[Watt93] David A. Watt
Programming Language Processors
Prentice Hall International Series in Computer Science. 1993

10.1 Función del intérprete

Un intérprete toma un programa fuente y lo ejecuta inmediatamente, esta es la clave de la interpretación, la no existencia de un tiempo anterior donde se ha traducido.

¿Cuándo es **útil** un intérprete?

- El programador trabaja en un entorno interactivo y se desean obtener los resultados de la ejecución de una instrucción antes de ejecutar la siguiente.
- El programador lo ejecuta escasas ocasiones y el tiempo de ejecución no es importante.
- Las instrucciones del lenguaje tiene una estructura simple y pueden ser analizadas fácilmente.
- Cada instrucción será ejecutada una sola vez.

10.1 Función del intérprete

¿Cuándo **no** es **útil** un intérprete?

- Si las instrucciones del lenguaje son complejas.
- Los programas van a trabajar en modo de producción y la velocidad es importante
- Las instrucciones serán ejecutadas con frecuencia.

10.2 Tipos de intérpretes

Interpretación iterativa

1. Comienzo
2. Repetir hasta la última instrucción:
   - Capturar instrucción
   - Analizar la instrucción
   - Ejecución
3. Fin

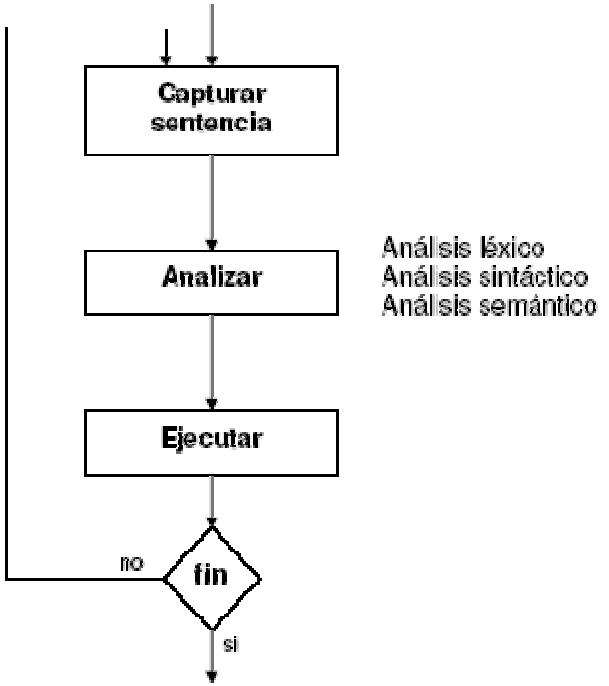
Interpretación recursiva

1. Comienzo
2. Captura y análisis
3. Ejecución:
   - Si la instrucción es compuesta: ejecución recursiva del intérprete
   - Si la instrucción es simple: se ejecuta la instrucción.
4. Fin

10.2.1 Intérpretes iterativos

Se realiza cuando todas las instrucciones del lenguaje fuente son primitivas (no son sentencias compuestas o estructuradas).

* Comienzo
* Repetir:
    * Capturar instrucción
    * Analizar la instrucción
    * Ejecutarla
* Hasta la ultima instrucción
* Fin



10.2.1.1 Intérprete iterativo para un lenguaje de órdenes

Descripción del lenguaje de órdenes:

script	::=	command*
command	::=	command_name argument* end_of_line
argument	::=	file_name
	|	literal
command_name	::=	**create** name
	|	**delete**
	|	**edit**
	|	**listfiles**
	|	**quit**
	|	file_name

Secuencia de órdenes:

```
delete f1 f2
create f3 100
listfiles
edit f3
sort f3
quit
```

10.2.1.1 Intérprete iterativo para un lenguaje de órdenes

Código del interprete:

```
/* inicio */

/* Inicialización … */
status := running ;
do {
    /* Captar la siguiente sentencia y analizarla … */
    fetchAndAnalyze (com) ;
    /* ejecuta la instrucción … */
    if (com.name == 'create')
       create (com.args[1], com.args[2]) ;
    else if (com.name == 'delete')
       delete (com.args) ;
    else if (com.name == 'edit')
       edit (com.args[1]) ;
    else if (com.name == 'listfiles')
       listFiles () ;
    else /* programa ejecutable */
       run (com.name, com.args) ;
} while (status != running) ; /* hasta que finalice la ejecución */

/* fin */
```

10.2.1.1 Intérprete iterativo para un lenguaje de órdenes

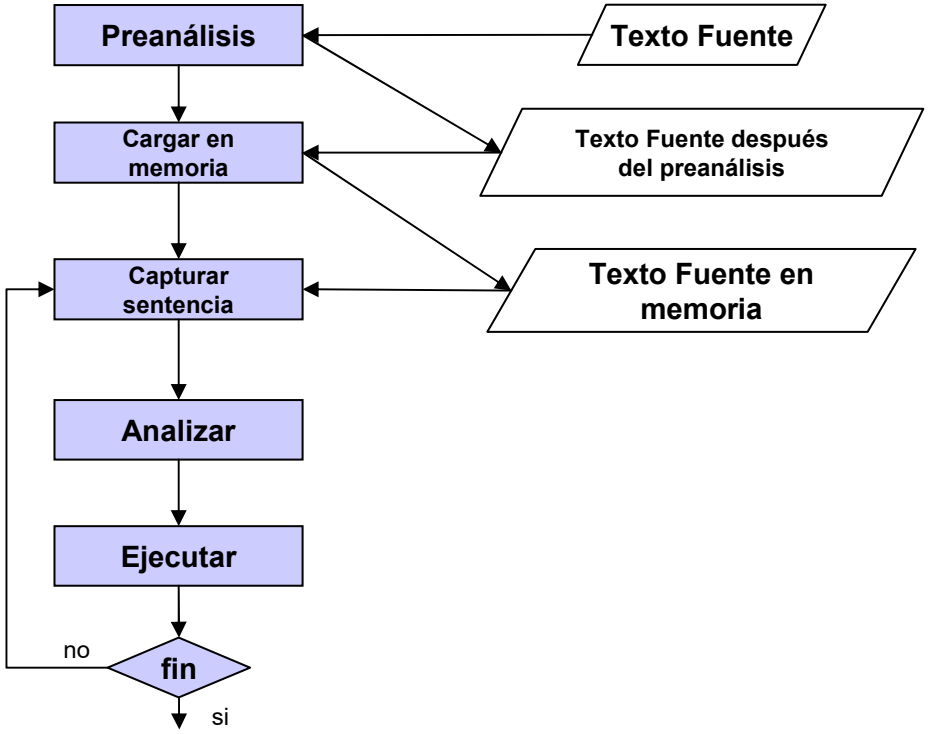
```
typedef char token[10] ;
typedef token tokensequence[8] ;
typedef struct {
    token name ;
    tokensequence args ;
} command ;
```

```
void fetchAndAnalyze (command com) ;          /* Lee y analiza la siguiente sentencia */
void create (token fname, token size) ;       /* Crea un archivo según el nombre y tamaño
                                                 dado por argumento */
void delete (tokensequence fnames) ;          /* Borra los archivos especificados por su nombre */
void edit (token fname) ;                     /* Edita el archivo especificado por su nombre */
void listFiles (void) ;                       /* Lista los nombres de todos los archivos propios
                                                 del usuario actual */
void run (token name, tokensequence args) ;   /* Ejecuta el programa en código máquina que contiene
                                                 la propia instrucción con los argumentos aportados
                                                 en ella */
```

```
typedef enum status {running, halted, failed } ;
command com ;
```

10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

El primer paso para la construcción del interprete es diseñar la máquina abstracta definida por el lenguaje.



10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

Reparto de las tareas de la fase de análisis

Preanálisis	Análisis
-----	léxico, sintáctico y semántico
léxico	sintáctico y semántico
léxico, sintáctico y semántico	-----

COMPROMISO ENTRE EDICIÓN DIRECTA Y EFICIENCIA

Los distintos formatos de las instrucciones en memoria son:

- ***Como texto.*** Esta opción implica que cada vez que una instrucción es capturada se ha de realizar una análisis de léxico, sintáctico (y semántico si es necesario). La carga y la edición es fácil.
- ***Como una secuencia de símbolos*** (componentes léxicas).Se realiza un análisis de léxico de las instrucciones antes de ser cargadas en memoria y serán analizadas sintáctica y semánticamente cuando son capturadas para ser ejecutadas. La carga y la edición son relativamente fácil de realizar.
- ***Como un árbol sintáctico***. En esta caso, la edición es difícil, por otro lado las instrucciones deben ser analizadas léxica y sintácticamente cuando son cargadas, pero estarán preparadas para una ejecución inmediata.

10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

DESCRIPCIÓN DE LENGUAJE (MINI BASIC)

program	::=	command *
command	::=	variable = expression
	|	**read** variable
	|	**write** variable
	|	**go** label
	|	**if** expression relational_op expresion **go** label
expresion	::=	primary_expression
	|	expression arithmetic_op primary_expression
primary_expression	::=	numeral
	|	variable
	|	(expression)
arithmetic_op	::=	'+' | '-' | '*' | '/'
relational_op	::=	'=' | '<' | '>' | '!='| '<='|'>='
numeral	::=	digito +
digito	::=	0|1|···|9
variable	::=	a|b|···|z

10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

Interpretación iterativa de un lenguaje de alto nivel sencillo

El primer paso para la construcción del intérprete es diseñar la máquina abstracta definida por el lenguaje.

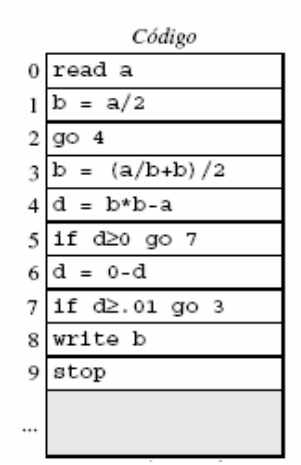
En el ejemplo usamos la siguiente máquina:

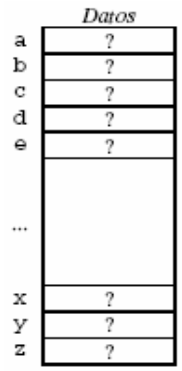
1. El programa es cargado en una zona de memoria dividida en celdas. Cada celda contiene una instrucción.
2. Contador de programa **CP**.
3. Los datos están almacenados en **26** celdas. Una para cada variable.

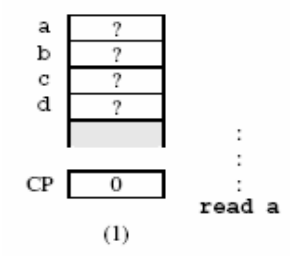
Respecto a las instrucciones hemos de diferenciar:

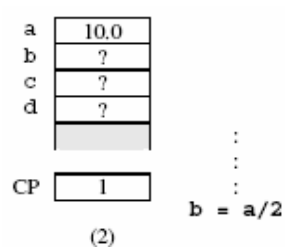
1. Su carga en memoria.
2. Su captura de memoria para ser ejecutadas.

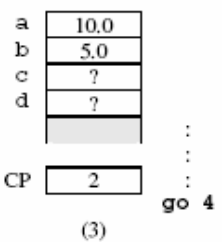
10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

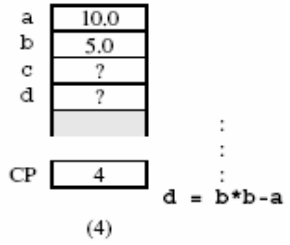


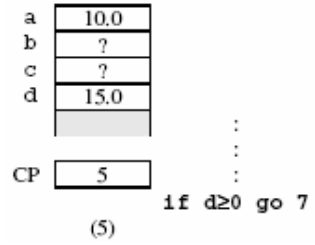


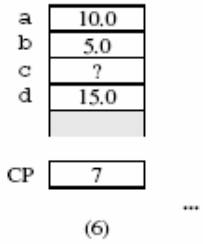












10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

Código del intérprete:

Datos

```
type Variable = 'a'..'z' ;
var  data : array [Variable] of Real ;
```

Código

```
type codeAddress = 0..maxCodeAddress ;
var  code : array [codeAddress] of StoredCommand ;
```

Sentencia

```
type Token = ... ;
     StoredCommand = array [1..12] of Token ;

var  status    : (running, halted, failed) ;
     CP        : CodeAddress ;
     storedCom : StoredCommand ;
     com       : Command ;
```

Representación de la sentencia después del análisis

```
type Expression = ... ;
     CommandKind = (VeqE, readV, writeV, goL, ifEREgoL, stop) ;
     Command = record
                 case kind : CommandKind of
                    VeqE :
                           ( lhs : Variable ;
                             rhs : Expression );
                    readV, writeV :
                           ( ioVar : Variable );
                    goL :
                           ( destination : CodeAddress );
                    ifEREgoL :
                           ( relop : Token ;
                             expr1, expr2 : Expression;
                             ifDestination : CodeAddress );
                    stop :
                           ( )
               end ;
```

Evalúa la expresión

```
function evaluate (expr : Expression) : Real ;
... ;
```

Evalúa una operación de relación

```
function compare (relop : Token; num1, num2 : Real)
                 : Boolean ;
...
```

Análisis Sintáctico y Semántico

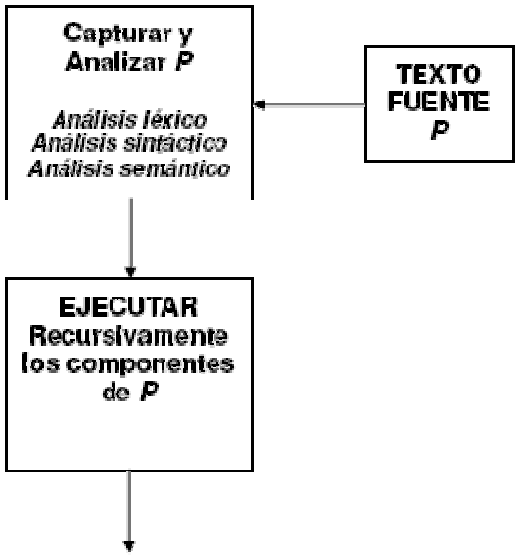
```
procedure analyze (storedCom : StoredCommand ;
                   var com : Command) ;
... { Analiza las sentencias almacenadas }
```

10.2.1.2 Intérprete iterativo para un lenguaje de alto nivel sencillo

```
begin
{ inicialización... }
status := running ;
CP := 0 ;

repeat

   { captar la siguiente sentencia... }
   storedCom := code[CP] ;
   CP := CP + 1 ;

   { analiza la instrucción... }
   analyze (storedCom, com) ;

   { ejecuta la instrucción... }
   case com.kind of
      VeqE     : data[com.lhs] := evaluate (com.rhs) ;

      readV    : readln (data[com.ioVar]) ;

      writeV   : writeln (data[com.ioVar]) ;

      goL      : CP := com.destination ;

      ifEREgoL : if compare (com.relop,
                             evaluate (com.expr1),
                             evaluate (com.expr2)) then
                    CP := com.ifDestination ;

      stop     : status := halted ;
   end { case }

   until status <> running   { ...finalice la ejecución }
end
```

10.2.2 Intérpretes recursivos

Es necesario si el lenguaje fuente tiene instrucciones o sentencias compuestas.



El esquema del interprete es el siguiente:

*** Comienzo**
*** Capturar y analizar C**

En esta parte se ha de realizar el análisis de C (léxico, sintáctico y semántico), esto conlleva a realizar el análisis de las partes de C.

*** Ejecutar C**

Operaría sobre la representación de C obtenida en el paso anterior (ej. Árbol sintáctico) y ejecutaría recursivamente las sub-instrucciones que componen C.

*** Fin**