NOTA Tanto para definir el léxico como para especificar la sintaxis de nuestro lenguaje Tiny, utilizamos la notación de Backus-Naur extendida (EBNF, por sus siglas en inglés: *Extended Backus-Naur Form*) definida por el estándar ISO/IEC 14977:1996.

1. Análisis léxico

A continuación se presenta el léxico de nuestro lenguaje Tiny, donde se consideran indistinguibles las letras mayúsculas (códigos *US-ASCII* del **65** al **90** [ambos inclusive]) de las minúsculas (códigos *US-ASCII* del **97** al **122** [ambos inclusive]), característica de algunos lenguajes de programación (p. ej.: Ada, BASIC, Fortran) denominada **case-insensitivity**:

```
(* 1.1 Caracteres *)

caracter = ? Carácter codificado en formato UTF-8 (ISO/IEC 10646) ?;

letra = ‘A’ | ‘B’ | ‘C’ | ‘D’ | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
      | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
      | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g'
      | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r'
      | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z';

dígito = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

cero símbolo = ‘0’;

dígito positivo = dígito - cero símbolo;

(* 1.2 Unidades léxicas *)

identificador = letra, { letra | dígito | subrayado };

entero = [ signo ], ( cero símbolo | dígito positivo, { dígito } );

real = entero, [ decimal | exponencial | decimal, exponencial ];

decimal = punto decimal, ( cero símbolo | { dígito }, dígito positivo );
exponencial = símbolo exponencial, entero;

cadena = delimitador cadena, { caracter imprimible }, delimitador cadena;

caracter imprimible = caracter - ( comilla | separador );

puntuacion = paréntesis de apertura | paréntesis de cierre
      | corchete de apertura | corchete de cierre
      | punto y coma
```

```
      | coma
      | dos puntos
      | punto;

operador = suma símbolo
      | resta símbolo
      | mul símbolo
      | div símbolo
      | mod símbolo
      | op relacional
      | asignacion
      | acc símbolo
      | indir símbolo

op relacional = eq símbolo | ne símbolo
      | le símbolo | ge símbolo
      | lt símbolo | gt símbolo;

palabras reservadas = tipo básico
      | and símbolo | or símbolo | not símbolo
      | nulo | booleano
      | proc símbolo | if símbolo | then símbolo | else símbolo
      | while símbolo | do símbolo
      | seq símbolo | begin símbolo | end símbolo
      | registro símbolo | array símbolo | of símbolo
      | new símbolo | delete símbolo
      | read símbolo | write símbolo | newline símbolo
      | var símbolo | tipo símbolo;

(* 1.3 Cadenas ignorables *)

separador = retroceso | retorno de carro | salto de línea;

blanco = ' ';

comentario = comentario símbolo, { caracter linea };

caracter linea = caracter - salto de línea;

(* 1.4 Símbolos terminales *)

subrayado = '_';
delimitador cadena = "'";
comentario símbolo = '@';

(* 1.4.1 Notación decimal *)
```

```
signo = ‘-’ | ‘+’;
punto decimal = '.';
símbolo exponencial = 'e' | 'E';

(* 1.4.2 Separadores *)

retroceso = '\b';
retorno de carro = '\r';
salto de línea = '\n';

(* 1.4.3 Puntuación *)

paréntesis de apertura = '(';
paréntesis de cierre =  ')';

corchete de apertura = '[';
corchete de cierre = ']';

punto y coma = ';';
coma = ',';
dos puntos = ':';
punto = ‘.’;

(* 1.4.4 Declaraciones *)

var símbolo = 'var';
tipo símbolo = 'type';
proc símbolo = 'proc';

(* 1.4.5 Tipos *)

tipo basico = 'int' | 'bool' | 'real' | 'string';

of símbolo = 'of';
array símbolo = 'array';
registro símbolo = 'record';
punt símbolo = ‘^’;

(* 1.4.6 Instrucciones *)

asignación = ‘=’;

if símbolo = ‘if’;
then símbolo = ‘then’;
else símbolo = ‘else’;
```

```
while símbolo = ‘while’;
do símbolo = ‘do’;

read símbolo = ‘read’;
write símbolo = ‘write’;
newline símbolo = ‘nl’;

new símbolo = ‘new’;
delete símbolo = ‘delete’;

seq símbolo = ‘seq’;
begin símbolo = 'begin';
end símbolo = 'end';

(* 1.4.7 Expresiones *)

booleano = 'true' | 'false';
nulo = ‘null’;

eq símbolo = '==';
ne símbolo = '!=';
le símbolo = '<=';
ge símbolo = '>=';
lt símbolo = '<';
gt símbolo = '>';

suma símbolo = '+';
resta símbolo = '-';
mul símbolo = '*';
div símbolo = '*';
mod símbolo = '%';

and símbolo = ‘and’;
or símbolo = ‘or’;
not símbolo = ‘not’;

indir símbolo = '^';
acc símbolo = '.';
```

2. Análisis sintáctico y construcción de ASTs (gramáticas S-atribuidas)

Por comodidad en la lectura, se muestran de forma intercalada las gramáticas incontextuales, que describen las **reglas sintácticas** del lenguaje Tiny, junto con las gramáticas de atributos que definen las acciones por las cuales se **construye** cada **nodo AST**, y por consiguiente, el

árbol de sintaxis abstracta. Estas últimas gramáticas se escriben empleando una notación muy parecida al BNF tradicional, por la utilización de los paréntesis angulares (<>) y los símbolos de definición (:=). Por otra parte, cualquier atributo (i. e.: incluyendo los atributos heredados, sh) de toda regla de la forma $X = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ es el de aquella α_i con la que se expande X, que hay una **como mucho una** debido a que la gramática no es ambigua:

```
(* 2.1 Programa *)

prog = sección decs, sección ins, punto;

<prog>.atr := prog(<sección decs>.atr, <sección ins>.atr)

sección decs = { dec, punto y coma };

<sección decs>.atr := { dec, punto y coma }.atr

sección ins = begin símbolo, { ins }, end símbolo;

<sección ins>.atr := { ins }.atr

(* 2.2 Declaraciones *)

dec = dec var | dec tipo | dec proc;

dec var = var símbolo, identificador, dos puntos, tipo;

<dec var>.atr := var(<identificador>, <tipo>.atr)

dec tipo = tipo símbolo, identificador, dos puntos, tipo;

<dec tipo>.atr := type(<identificador>, <tipo>.atr)

dec proc = proc símbolo, identificador, paréntesis apertura,
      pformales, paréntesis cierre, sección decs, sección ins;

<dec proc>.atr :=
      proc(identificador, <pformales>.atr, { decs }.atr, { ins }.atr)

(* 2.3 Parámetros formales *)

pformales = [ { pformal,  coma }, pformal ];

<pformales>.atr := lparam([ { pformal, coma }, pformal ].atr)

lparam(opc_indefinido()): no_params()
```

```
lparam(opc_definido({ pformal₀, coma }, pformal₁)):
      varios_params({ pformal₀, coma }.atr, pformal₁.atr)

pformal = [ var símbolo ], identificador, dos puntos, tipo;

<pformal>.atr := param(<identificador>, <tipo>.atr, [ var símbolo ].atr)

(* 2.4 Tipos *)

tipo = tipo básico (* Definido en "Tipos" [1.4.5] *)
      | tipo renombrado
      | tipo array
      | tipo registro
      | tipo puntero;

tipo renombrado = identificador;

<tipo renombrado>.atr := ref(<identificador>)

(* 2.5 Tipos compuestos *)

tipo array = array símbolo, corchete apertura, entero, corchete cierre,
      of símbolo, tipo;

<tipo array>.atr := array(entero, <tipo>.atr)

tipo puntero = punt símbolo, tipo;

<tipo puntero>.atr := puntero(<tipo>.atr)

(* 2.6 Registros *)

tipo registro = registro símbolo, { campo }-, end símbolo;

<tipo registro>.atr := registro({ campo }-.atr)

campo = identificador, dos puntos, tipo, punto y coma;

<campo>.atr := campo(<identificador>, <tipo>.atr)

(* 2.7 Instrucciones *)

ins = ins asig
      | ins condicional
      | ins while
      | ins lectura
```

```
      | ins escritura
      | ins newline
      | ins new
      | ins delete
      | ins invoc
      | ins compuesta; (* 1 *)

ins asig = expresión, asig símbolo, expresión, punto y coma; (* 2 *)

<ins asig>.atr := asig(<expresión>0.atr, <expresión>1.atr)

ins invoc = expresión, paréntesis apertura, preales,
      paréntesis cierre, punto y coma; (* 3 *)

<ins invoc>.atr := invoc(<expresion>.atr, <preales>.atr)

(* 2.8 Bloques *)

ins condicional = if símbolo, expresión, then símbolo, { ins },
      [ else símbolo, { ins } ], end símbolo, [ punto y coma ];

<ins condicional>.atr := ins_condicional(<expresión>.atr, { ins }0.atr,
      [ else símbolo, { ins }1 ].atr)

ins while = while símbolo, expresión, do símbolo, { ins }, end símbolo,
      [ punto y coma ];

<ins while>.atr := while(<expresión>.atr, { ins }.atr)

ins compuesta = seq símbolo, sección decs, sección ins, [ punto y coma ];

<ins compuesta>.atr := seq(<sección decs>.atr, <sección ins>.atr)

(* 2.9 Gestión de E/S *)

ins lectura = read símbolo, expresión, punto y coma;

<ins lectura>.atr := read(<expresión>.atr)

ins escritura = write símbolo, expresión, punto y coma;

<ins escritura>.atr := write(<expresion>.atr)

ins newline = newline símbolo, punto y coma;
```

```
<ins newline>.atr := nl()

(* 2.10 Gestión de memoria dinámica *)

ins new = new símbolo, expresión, punto y coma;

<ins new>.atr := new(<expresion>.atr)

ins delete = delete símbolo, expresión, punto y coma;

<ins delete>.atr = delete(<expresión>.atr)

(* 2.11 Parámetros reales *)

preales = [ preal, { coma, preal } ];

<preales>.atr := lreal([ preal0, { coma, preal1 } ].atr)

lreal(opc_indefinido()): no_exp()
lreal(opc_definido({ preal0, coma }, preal1)):
      varias_exp({ preal0, coma }.atr, preal1.atr)

preal = expresión;

<preal>.atr := <expresión>.atr

(* 2.12 Expresiones *)

expresión = exp nivel 0;

(* 2.13 Expresiones compuestas *)

exp nivel 0 = exp binaria 0 | exp nivel 1;

exp binaria 0 = (* Op. binario, infijo, no asociativo *)
      exp nivel 1,
      op relacional, (* Definido en "Unidades léxicas" [1.2] *)
      exp nivel 1;

<exp binaria 0>.atr :=
      op_binario(<exp nivel 1>0.atr, <exp nivel 1>1.atr, <op relacional>)

exp nivel 1 = más binario | menos binario
      | exp nivel 2; (* 4 *)
```

```
más binario = (* Op. binaria, no asociativa *)
      exp nivel 2, suma, exp nivel 2;

<más unario>.atr := <exp nivel 2>.atr

menos binario = (* Op. binario, asociativo a izquierdas *)
      exp nivel 1, resta, exp nivel 2;

<menos binario>.atr := resta(<exp nivel 1>.atr, <exp nivel 2>.atr)

exp nivel 2 = and | or
      | exp nivel 3; (* 5 *)

and = (* Op. binario, no asociativo *)
      exp nivel 3, and símbolo, exp nivel 3; (* 6 *)

<and>.atr := and(<exp nivel 3>0.atr, <exp nivel 3>1.atr)

or = (* Op. binario, asociativo a derechas *)
      exp nivel 3, or símbolo, exp nivel 2; (* 7 *)

<or>.atr := or(<exp nivel 3>.atr, <exp nivel 2>.atr)

exp nivel 3 = exp binaria 3 | exp nivel 4;

exp binaria 3 = (* Op. binario, infijo, asociativo a izquierdas *)
      exp nivel 3, op nivel 3, exp nivel 4; (* 8 *)

<exp binaria 3>.atr := op_binario(<exp nivel 3>.atr, <exp nivel 4>.atr,
      <op nivel 3>)

op nivel 3 = mul símbolo | div símbolo | mod símbolo;

exp nivel 4 = exp unaria 4 | exp nivel 5;

exp unaria 4 = (* Op. unario, prefijo, asociativo *)
      op nivel 4, exp nivel 4;

<exp nivel 4>.atr := op_unario(<exp nivel 4>.atr, <op nivel 4>)

op nivel 4 = resta símbolo | not símbolo;

exp nivel 5 = exp parentizada
      | exp posfija
      | exp básica; (* 9 *)
```

```
exp parentizada = paréntesis apertura, exp nivel 0, paréntesis cierre;

<exp parentizada>.atr := <exp nivel 0>.atr

exp básica = entero | real
      | cadena | identificador
      | booleano | null símbolo;

(* 2.14 Operadores posfijos *)

exp posfija = indx | acc | indir;

indx = (* Op. unario, posfijo, asociativo *)
      exp nivel 5, corchete apertura, expresión, corchete cierre; (* 10 *)

<indx>.atr := indx(<exp nivel 5>.atr, <expresión>.atr)

acc = exp nivel 5, acc símbolo, identificador; (* 11 *)

<acc>.atr := acc(<exp nivel 5>.atr, <identificador>)

indir = exp nivel 5, indir símbolo; (* 12 *)

<indir>.atr := indir(<exp nivel 5>.atr)
```

2.1. Gramática de atributos: Funciones auxiliares

Esta sección define las funciones y definiciones auxiliares empleadas en la descripción de la sintaxis de Tiny. Para ello, se utiliza la **semántica** de la notación EBNF, donde las llaves reflejan la **repetición** de una secuencia de símbolos terminales y no terminales y los corchetes describen una secuencia **opcional**. Estas definiciones vienen dadas por las siguientes reglas **genéricas** en notación EBNF, donde el tipo genérico se corresponde con la variable (i. e.: símbolo no terminal) que aparece entre los paréntesis angulares (<>) y dichas reglas son reificadas siguiendo el procedimiento de instanciación de plantillas (templates) de C++:

NOTA Los símbolos *wildcard* (_) solo sirven para denotar cualquier concatenación, posiblemente vacía, de terminales y no terminales a los lados de una expresión determinada:

```
(* 2.1.1 Símbolos terminales *)

cadena vacía símbolo = ''

(* 2.1.2 Cierre de Kleene *)
```

```
(* Secuencia repetida [ver sec. 5.6 de la norma ISO de EBNF] *)

{ T }.atr := { _, T }.atr := { T, _ }.atr := { _, T, _ }.atr :=
      lista<T>.atr

lista<T> = no_elems<T> | varios_elems<T>;

no_elems<T> = cadena vacía símbolo;

no_elems<T>.atr := lista_vacía(T)

varios_elems<T> = lista<T>, e;

varios_elems<T>.atr := lista_no_vacía(T, lista<T>.atr, e.atr)

(* 2.1.3 Cierre de Kleene positivo *)

{ T }-.atr := { _, T }-.atr := { T, _ }-.atr := { _, T, _ }-.atr :=
      lista_no_vacía<T>.atr

lista_no_vacía<T> = un_elem<T> | dos_o_más_elems<T>;

un_elem<T> = e;

un_elem<T>.atr := lista unitaria(T, e.atr)

dos_o_más_elems<T> = lista_no_vacía<T>, e;

dos_o_más_elems<T>.atr := lista_no_vacía(T, lista<T>.atr, e.atr)

(* 2.1.4 Secuencia opcional *)

(* Ver definición en sec. 5.5 de la norma ISO de EBNF *)

[ T ].atr := [ _, T ].atr := [ T, _ ].atr := [ _, T, _ ].atr :=
      opcional<T>.atr

opcional<T> = indefinido<T> | definido<T>;

indefinido<T> = cadena vacía símbolo;

indefinido<T>.atr = opc_indefinido()

definido<T> = e;

definido<T>.atr = opc_definido(T)
```

2.1.5 Constructores auxiliares de nodos AST

```
// Listas

lista_vacía(dec) := no_decs()
lista_no_vacía(dec, Ds, D) := varias_decs(Ds, D)

lista_vacía(ins) := no_ins()
lista_no_vacía(ins, Is, I) := varias_ins()

lista_vacía(param) := no_params()
lista_no_vacía(param, PFs, PF) := varios_params(PFs, PF)

lista_unitaria(campo, C) := un_campo(C)
lista_no_vacía(campo, Cs, C) := varios_campos(Cs, C)

lista_vacía(preales) := no_exp()
lista_no_vacía(preales, PRs, PR) := varias_exps(PRs, PR)

// Parámetros formales

param(id, T, opc_indefinido()) := pval(id, T)
param(id, T, opc_definido(param)) := pvar(id, T)

// Instrucción condicional

ins_condicional(E, Is, opc_indefinido()) := if-then(E, Is)
ins_condicional(E, Is, opc_definido(Ie)) := if-then-else(E, Is, Ie)

// Comparadores

op_binario(op1, op2, <eq símbolo>) := eq(op1, op2)
op_binario(op1, op2, <ne símbolo>) := ne(op1, op2)
op_binario(op1, op2, <le símbolo>) := le(op1, op2)
op_binario(op1, op2, <lt símbolo>) := lt(op1, op2)
op_binario(op1, op2, <ge símbolo>) := ge(op1, op2)
op_binario(op1, op2, <gt símbolo>) := gt(op1, op2)

// Operadores aritméticos

op_binario(op1, op2, <mul símbolo>) := mul(op1, op2)
op_binario(op1, op2, <mod símbolo>) := mod(op1, op2)

// Operadores unarios
```

```
op_unario(op, <resta símbolo>) := neg(op)
op_unario(op, <not símbolo>) := not(op)
```

3. Acondicionamiento de la especificación

En esta sección acondicionamos algunos aspectos definidos en secciones anteriores para facilitar una interpretación eficiente y precisa de la implementación descendente del constructor del AST.

En concreto, necesitamos que la gramática incontextual de la sección 2 sea del tipo **LL(*k*)**, por lo que debe ser no ambigua y carecer de recursión y factores comunes a izquierdas. Dicha recursión, si requiere solo un paso de derivación para manifestarse (i. e.: es directa), se puede eliminar utilizando el **lema de Arden**, por el cual una ecuación de lenguajes de la forma $L = A\,L \cup B$ tiene como solución $L = A * B$, donde el asterisco denota el cierre de Kleene.

A continuación, mostramos las reglas, junto con la ecuación característica asociada, que han requerido dichas transformaciones para convertir la gramática incontextual en una que no presenta factores comunes a izquierdas (reglas de la **1** a la **3**, de la **5** a la **7** y de la **10** a la **13** [ambos inclusive]) ni recursión a izquierdas (todas las reglas):

```
(* 1 *) ins = ins ident
      | ins condicional
      | ins while
      | ins lectura
      | ins escritura
      | ins newline
      | ins new
      | ins delete
      | ins compuesta;

ins ident = expresión, resto ident;

<resto ident>.sh := <expresión>.atr
<ins ident>.atr := <resto ident>.atr

resto ident = ins asig | ins invoc;

(* 2 *) ins asig = asig símbolo, expresión, punto y coma;

<ins asig>.atr := asig(<resto ident>.sh, <expresión>.atr)

(* 3 *) ins invoc = paréntesis apertura, preales, paréntesis cierre, punto
y coma;
```

<ins invoc>.atr := ***invoc***(<*resto* ident>.sh, <preales>.atr)

(* 4 *) **exp nivel 1** = exp nivel 2, *opc* exp nivel 1, *resto* exp nivel 1;

<*opc* exp nivel 1>.sh := <exp nivel 2>.atr
<*resto* exp nivel 1>.sh := <*opc* exp nivel 1>.atr
<exp nivel 1>.atr := <*resto* exp nivel 1>.atr

opc exp nivel 1 = suma, exp nivel 2;

<*opc* exp nivel 1>.atr := ***suma***(<*opc* exp nivel 1>.sh, <exp nivel 2>.atr)

opc exp nivel 1 = cadena vacía símbolo;

<*opc* exp nivel 1>.atr := <*opc* exp nivel 1>.sh

resto exp nivel 1 = resta símbolo, exp nivel 2, *resto* exp nivel 1;

<*resto* exp nivel $1>_1$.sh := ***resta***(<*resto* exp nivel $1>_0$.sh, <exp nivel 2>.atr)
<*resto* exp nivel $1>_0$.atr := <*resto* exp nivel $1>_1$.atr

resto exp nivel 1 = cadena vacía símbolo;

<*resto* exp nivel 1>.atr := <*resto* exp nivel 1>.sh

(* 5 *) **exp nivel 2** = exp nivel 3, op nivel 2;

<op nivel 2>.sh := <exp nivel 3>.atr
<exp nivel 2>.atr := <op nivel 2>.atr

op nivel 2 = cadena vacía símbolo;

<op nivel 2>.atr := <op nivel 2>.sh

op nivel 2 = and;

<and>.sh := <op nivel 2>.sh
<op nivel 2>.atr := <and>.atr

op nivel 2 = or;

<or>.sh := <op nivel 2>.sh
<op nivel 2>.atr := <or>.atr

(* 6 *) **and** = and símbolo, exp nivel 3;

<and>.atr := ***and***(<and>.sh, <exp nivel 3>.atr)

(* 7 *) **or** = or símbolo, exp nivel 2;

<or>.atr := ***or***(<or>.sh, <exp nivel 2>.atr)

(* 8 *) **exp binaria 3** = exp nivel 4, *resto* exp binaria 3;

<*resto* exp binaria 3>.sh := <exp nivel 4>.atr
<exp binaria 3>.atr := <*resto* exp binaria 3>.atr

resto exp binaria 3 = cadena vacía;

<*resto* exp binaria 3>.atr := <resto exp binaria 3>.sh

resto exp binaria 3 = mul símbolo, exp nivel 4, *resto* exp binaria 3;

<*resto* exp binaria $3>_1$.sh := ***mul***(<*resto* exp binaria $3>_0$.sh, <exp nivel 4>.atr)
<*resto* exp binaria $3>_0$.atr := <*resto* exp binaria $3>_1$.atr

resto exp binaria 3 = div símbolo, exp nivel 4, *resto* exp binaria 3;

<*resto* exp binaria $3>_1$.sh :=
    div(<*resto* exp binaria $3>_0$.sh, <exp nivel 4>.atr)
<*resto* exp binaria $3>_0$.atr := <*resto* exp binaria $3>_1$.atr

resto exp binaria 3 = mod símbolo, exp nivel 4, *resto* exp binaria 3;

<*resto* exp binaria $3>_1$.atr :=
    mod(<*resto* exp binaria $3>_0$.sh, <exp nivel 4>.atr)

<*resto* exp binaria $3>_1$.sh := *resto* exp binaria 3_1.atr

(* 9 *) **exp nivel 5** = exp no posfija, *resto* exp posfija;

<*resto* exp posfija>.sh = <exp no posfija>.atr
<exp nivel 5>.atr := <*resto* exp posfija>.atr

resto exp posfija = cadena vacía símbolo;

<***resto*** exp posfija>.atr = <*resto* exp posfija>.sh

resto exp posfija = indx, *resto* exp posfija;

<indx>.sh = <*resto* exp posfija>$_0$.sh

<*resto* exp posfija>$_1$.sh = indx.atr
<*resto* exp posfija>$_0$.atr = <*resto* exp posfija>$_1$.atr

resto exp posfija = acc, *resto* exp posfija;

<acc>.sh = <*resto* exp posfija>$_0$.sh
<*resto* exp posfija>$_1$.sh = <acc>.atr
<*resto* exp posfija>$_0$.atr = <*resto* exp posfija>$_1$.atr

resto exp posfija = indir, *resto* exp posfija;

<indir>.sh = <*resto* exp posfija>$_0$.sh
<*resto* exp posfija>$_1$.sh = <indir>.atr
<*resto* exp posfija>$_0$.atr = <*resto* exp posfija>$_1$.atr

exp no posfija = exp parentizada | exp básica;

(* 10 *) **indx** = corchete apertura, expresión, corchete cierre;

<indx>.atr = ***indx***(<indx>.sh, <expresion>.atr)

(* 11 *) **acc** = acc símbolo, identificador;

<acc>.atr = ***acc***(<acc>.sh, <identificador>.lex)

(* 12 *) **indir** = indir símbolo;

<indir>.atr := indir(<indir>.sh)