Autómatas y Lenguajes

3er curso

1er cuatrimestre

Alfonso Ortega: alfonso.ortega@uam.es

UNIDAD 2: Procesadores de lenguaje

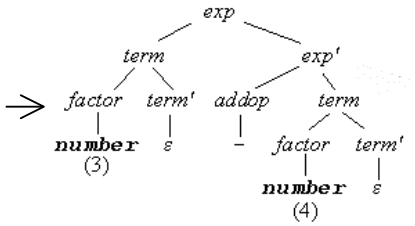
TEMA 5: Análisis Morfológico

Tema 5: Analizador morfológico

5.1

Introducción

Introducción (I)




```
begin
   int A;

   A := 100;
   A := A+A;
   output A
end
```

AM

```
(<palabra clave>,begin)
   (<tipo>,int)(<id>,A)(<simb>,;)

   (<id>,A)(<simbm>,:=)
      (<cons int>,100)(<simb>,;)
   (<id>,A)(<simbm>,:=)
      (<id>,A)(<simb>,+)(<id>,A)(<simb>,;)
   (<palabra clave>,output)(<id>,A)
(palabra clave>,end)
```


Introducción (II)

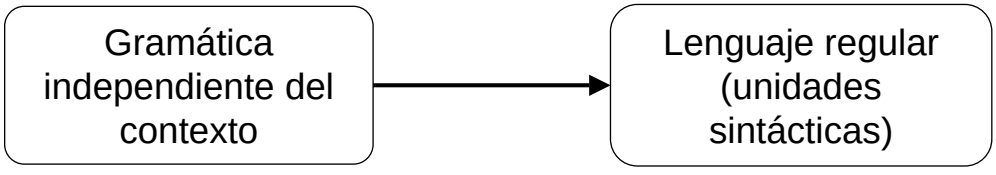
- También se le puede llamar scanner o analizador léxico.
- Su principal función es traducir el código fuente a una secuencia de unidades sintácticas:
  - Habitualmente identificadores, palabras reservadas, separadores, símbolos simples o múltiples y constantes.
  - Se pueden tratar como expresiones regulares.
  - Serán los símbolos terminales de la gramática usada por el analizador sintáctico.
- Otras tareas son:
  - Detección de errores morfológicos: por ejemplo identificadores que empiezan por un número y la gramática no lo permite.
  - Eliminación de blancos: normalmente los espacios en blanco, caracteres de tabulación o saltos de línea no proporcionan información y por lo tanto se pueden eliminar (con algunas excepciones como en Python)
  - Ignorar comentarios que son irrelevantes para la ejecución del programa.
  - Iniciar tareas del análisis semántico: calcular valor de las constantes, almacenar identificadores, etc.

5.2

Objetivos y conceptos previos

Objetivo

- Para la principal tarea del analizador morfológico, esto es, dividir el programa fuente en unidades sintácticas, es necesaria la identificación de las unidades sintácticas.
- Para ello, se analiza la gramática independiente del contexto del lenguaje de origen y se obtiene el conjunto de reglas que describen el lenguaje regular de las unidades sintácticas**,** buscándose una solución de compromiso
  - Entre el esfuerzo necesario y las ventajas conseguidas.
  - Que mantenga la corrección teórica.



Conceptos previos (I)

¿Qué se puede identificar con el analizador morfológico?

- Se pueden identificar palabras reservadas, como por ejemplo símbolos. Las reglas generales son:
  - Un símbolo es una secuencia de caracteres que no se puede separar por un espacio en blanco.
  - Los símbolos se separan entre ellos mediante espacios en blanco.
- Por ejemplo en la sentencia: A := 1 + 2
  - ':=' se puede separar de A y de 1
  - pero ':' no se puede separar de '='
  - Por lo tanto ':=' debería tratarse como un símbolo.
- En la gramática del lenguaje ASPLE, vamos a resaltar elementos que son frecuentemente identificados por el analizador morfológico:
  - ***Palabras reservadas***
  - ***Constantes numéricas***
  - ***Identificadores***
  - ***Constantes de tipo lógico***

Conceptos previos (II)

Unidades sintácticas de la gramática del lenguaje

```
1  :   <program>     ::=   begin <dcl train> ; <stm train> end
2  :   <dcl train>   ::=   <declaration>
3  :                 |     <declaration> ; <dcl train>
4  :   <stm train>   ::=   <statement>
5  :                 |     <statement> ; <stm train>
6  :   <declaration>::=    <mode> <idlist>
7  :   <mode>        ::=   bool
8  :                 |     int
9  :                 |     ref <mode>
10 :   <idlist>      ::=   <id>
11 :                 |     <id> , <idlist>
12 :   <statement>   ::=   <asgt stm>
13 :                 |     <cond stm>
14 :                 |     <loop stm>
15 :                 |     <transput stm>
15 :                 |     <case stm>
16 :                 |     call <id>
17 :   <asgt stm>    ::=   <id> := <exp>
18 :   <cond stm>    ::=   if <exp> then <stm train> fi
19 :                 |     if <exp> then <stm train> else <stm
 train> fi
```

Conceptos previos (III)

Unidades sintácticas de la gramática del lenguaje

```
20 :   <loop stm>    ::=   while <exp> do <stm train> end
21 :                 |     repeat <stm train> until <exp>
22 :   <transput stm> ::=    input <id>
23 :                  |      output <exp>
24 :   <exp>         ::=   <factor>
25 :                 |     <exp> + <factor>
26 :                 |     <exp> - <factor>
27 :                 |     - <exp>
28 :   <factor>      ::=   <primary>
29 :                 |     <factor> * <primary>
30 :   <primary>     ::=   <id>
31 :                 |     <constant>
32 :                 |     ( <exp> )
33 :                 |     ( <compare> )
34 :   <compare>     ::=   <exp> = <exp>
35 :                 |     <exp> <= <exp>
36 :                 |     <exp> > <exp>
```


Conceptos previos (IV)

Unidades sintácticas de la gramática del lenguaje

```
37 :   <constant>    ::=   <bool constant>
38 :                 |     <int constant>
39 :   <bool constant> ::=  true
40 :                  |     false
41 :   <int constant> ::=   <number>
42 :   <number>      ::=    <digit>
43 :                 |      <number> <digit>
44 :   <id>          ::=    <letter>
45 :                 |      <letter><rest id>
46 :   <rest id>     ::=    <alfanumeric>
47 :                 |      <alfanumeric><rest id>
46 :   <digit>       ::=    0 | 1 | ... | 9
47 :   <letter>      ::=    a | b | ... | z | A | B | ... | Z
48 :   <case stm>   ::=    case ( <expr> ) <constant case train> esac
49 :   <constant case train> ::=  <constant case>
50 :                          |    <constant case> <constant case train>
51 :   <constant case>        ::=  <int constant> : <stm train>
52 :   <alfanumeric>  ::= <digit>
53 :                  |   <letter>
54 :   <procedures>   ::= <procedure> <procedures>
55 :                  |
56 :   <procedure>    ::= procedure <id> begin <stm train> end
```

Conceptos previos (V)

Unidades sintácticas de la gramática del lenguaje

- La pregunta es ¿dónde parar?
  - En este caso, es fácil que el alumno detecte algunas reglas de producción en la gramática que son regulares, como por ejemplo:

```
7  :   <mode>        ::=   bool
8  :                |     int
9  :                |     ref <mode>
```

- La respuesta consiste en respetar el compromiso entre esfuerzo y beneficios.
- Tenga en cuenta los siguientes factores:
  - El alumno conoce, de la asignatura TALF, métodos para demostrar si un lenguaje es regular(*)
  - Tras un estudio más o menos costoso, se podría determinar, para cada lenguaje de programación, cuál es su sublenguaje regular
  - La mayoría de los lenguajes de programación tendrán **identificadores**, **palabras reservadas**, (incluidos símbolos especiales) y **constantes** que
    - Tienen una sintaxis que **varía poco** de un lenguaje de programación a otro.
    - Son un **lenguaje regular**.

Conceptos previos (VI)

Unidades sintácticas de la gramática del lenguaje

- Centrarse, como objetivo para el analizador morfológico, en el reconocimiento de
  - Identificadores
  - Constantes
  - Palabras reservadas
- Parece constituir un buen objetivo ya que
  - La repetición de su sintaxis en la mayoría de los casos, facilita su especificación correcta.
  - Permite, con poco esfuerzo, proporcionar al analizador morfológico una parte razonable del proceso de análisis.

Tipos de analizador morfológico (I)

De una o más pasadas

- Se puede distinguir entre:
  - Analizador morfológico de una pasada que identifica los símbolos e immediatamente asigna una etiqueta a cada símbolo.

```
"begin
   int A;
   A := 100;
   A := A+A;
   print A
end"
```

```
(begin,begin)(type, int) (id,A)
(semic,;) (id,A) (eqsgn,:=)
(int,100)(semic,;) (id,A)
(eqsgn,:=) (id,A) (symb,+) (id,A)
(semic,;)
(reserved-word,output) (id,A)
(end,end)
```

  - Analizador morfológico de más de una pasada. Ejemplo dos pasadas:
    - En la primera pasada identifica los símbolos.
    - En la segunda pasada asigna una etiqueta a cada símbolo.

Tipos de analizador morfológico (II)

Ejemplo de dos pasadas

- En un analizador morfológico de dos pasadas:
  - En la primera pasada se agrupan los caracteres en símbolos:

```
"begin
   int A;
   A := 100;
   A := A+A;
   print A
end"
```

"begin" "int", "A" ";" "A" ":=" "100" ";" "A" ":=" "A" "+" "A" ";" "print" "A" "end"

  - En la segunda pasada se asignan las etiquetas a cada símbolo:

"begin" "int", "A" ";" "A" ":=" "100" ";" "A" ":=" "A" "+" "A" ";" "print" "A" "end"

```
(begin,begin)(type, int) (id,A)
(semic,;) (id,A) (eqsgn,:=)
(int,100)(semic,;) (id,A)
(eqsgn,:=) (id,A) (symb,+) (id,A)
(semic,;)
(reserved-word,output) (id,A)
(end,end)
```

5.3

Implementación

Fases en la construcción del analizador morfológico

Definición de objetivos e implementación

- Definición de los objetivos del compilador:
  - Identificación de las unidades sintácticas: selección de la parte regular dentro de la gramática independiente del contexto del lenguaje completo.
  - Descripción de las acciones que el analizador debe realizar mientras reconoce las unidades sintácticas.
- Implementación del analizador
  - Mediante programación directa (“ad hoc”): se escribe código para reconocer cada token.
  - Utilizando herramientas formales
    - Representación formal de las unidades sintácticas
      - Autómata finito
      - Expresiones regulares
      - Gramática regular
    - Obtención del autómata finito equivalente a partir de la representación formal
      - Desde las expresiones regulares
      - Desde la gramática regular
    - Implementación del autómata finito
  - Utilización de herramientas de generación de analizadores

5.3.1

Programación directa

Programación directa (I)

Código ad-hoc

- La implantación de un analizador morfológico puede ser abordada como el desarrollo de una aplicación informática más:
  - La especificación de sus requisitos consiste en:
    - Describir el formato y contenido de las entradas (fuentes con o sin errores)
    - Describir el proceso (la identificación de la lista de unidades sintácticas presentes en el fichero fuente, comportamiento ante espacios en blanco, comentarios y errores morfológicos)
    - Describir la salida (el "token" siguiente o la secuencia de unidades sintácticas asociadas al fuente de entrada si es correcto, los mensajes de error en otro caso, etc.)
  - El diseño y codificación se realizan para poder reconocer cada una de las unidades sintácticas del lenguaje fuente. Ejemplo de código:

```
def tokenise():
    symbolList = []
    while not eof():
            // process next chars until end of symbol
            // add symbol to symbolList. . .
    return symbolList
```

Programación directa (II)

Reconociendo los símbolos

```
def tokenise():
    symbolList = []
    while not eof():
        case type(nextc):
         'whitespace':   ...
         'alpha':        ...
         'digit':        ...
         etc.

    return symbolList
```


Programación directa (III)

```
def tokenise():
    symbolList = []
    while not eof():
        case type(nextc):
         'whitespace':   ...
         'alpha':        ...
         'digit':        ...
         etc.

    return symbolList


def type (char):
   if char in “a-zA-z_”: return ‘alpha’
   if char in “0-9”:     return ‘digit’
   if char in “ \t\n”:   return ‘whitespace’
   if char in “{};,”:    return ‘sepchar’
   if char in “><=+-/*”:  return ‘mathchar’
```

Programación directa (IV)

```
def tokenise():
    symbolList = []
    while not eof():
        case type(nextc):

            'alpha': // alpha includes here '_'
                     symbol = “” + getc()
                     while type(nextc) in ['alpha', 'digit']:
                         symbol += getc()
                     symbolList.append(symbol)

            'whitespace':  getc()

            'digit':         ...

            ...
```


Programación directa (V)

```
def tokenise():
    symbolList = []
    while not eof():
        case type(nextc):

            'alpha': // alpha includes here '_'
                     symbol = “”+getc()
                     while type(nextc) in ['alpha', 'digit']:
                         symbol += getc()
                     symbolList.append(symbol)

            'whitespace':  getc()

            'digit':         ...

            ...
```

Programación directa (VI)

```
. . .

‘mathchar':   // = > < + - * /
                 symbol = “” + getc()
                 if nextc == '=':
                    symbol += getc()
                 symbolList.append(symbol)

'sepchar':     //   { } ; ,
               symbol = “” + getc()
               symbolList.append(symbol)

default:     print "ERROR: Unknown Char: “+getc()
```


Programación directa (VII)

Números:

- Formatos: 1, 34, 34.001, .0
- Procedimiento
  1) Leer dígitos hasta que se encuentre un no-dígito.
  2) Si el siguiente caracter es un “.” entonces seguir leyendo hasta no-dígito.

```
‘digit': symbol = “”+getc()
         while nextc in “0123456789”:
            symbol += getc()
         if nextc == “.”:
            symbol += getc()
            while nextc in “0123456789”:
               symbol += getc()
         symbolList.append(symbol)
```

Programación directa (VIII)

Asignando las etiquetas

- Se asignan las etiquetas a los símbolos identificados:

1. En el caso de las palabras reservadas por comparación o buscando en una tabla hash:

```
If Symbol in RESERVED_WORDS: Token = symbol
```

2. Para el resto de unidades sintácticas se procede de manera similar

Programación directa (IX)

Ejemplo de código ad-hoc para un analizador de un paso (identificar y etiquetar)

```
def tokenise():
    symbolList = []
    while not eof():
       while nextc in WHITE_SPACE_CHARS: getc()
       symbtype = type(nextc)
       symbol = “”+getc()
       case symbtype:
          'alpha': // alpha includes here '_'
              while type(nextc) in ['alpha', 'digit']:
                  symbol += getc()
                  
       ...
        symbolList.append( [token,symbol])
    return symbolList
```

5.3.2

Uso de herramientas formales (Expresiones y Gramáticas Regulares)

Uso de herramientas formales (I)

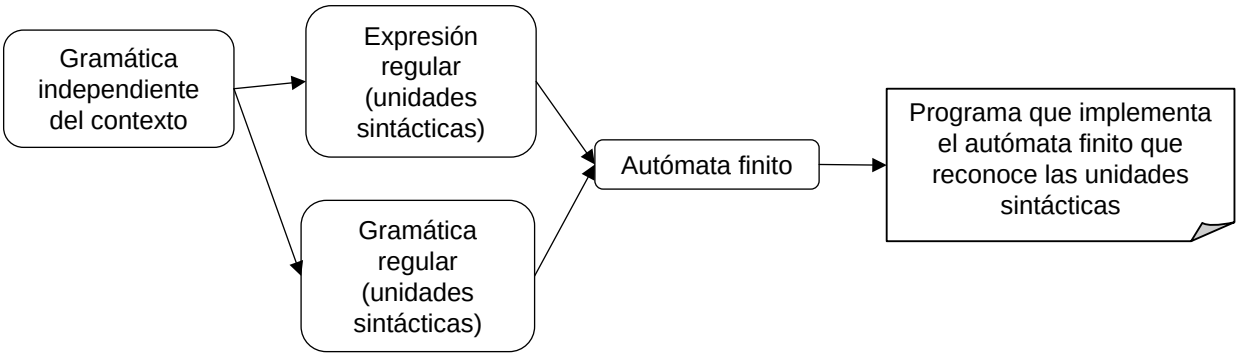
Alternativas para la construcción del analizador morfológico (I)

- El método de implementación del analizador morfológico estudiado en la sección anterior es informal, siendo necesario escribir a mano un programa específico capaz de reconocer y etiquetar las unidades sintácticas.
- Las reglas del análisis morfológico son implícitas, y se hace necesario leerse el código para poder encontrarlas.
- En programas muy básicos esto podría ser suficiente.
- Sin embargo, en los programas que actualmente se están usando, este enfoque es muy problemático:
  - Problemas de portabilidad: un cambio en la sintaxis del lenguaje requiere editar el analizador morfológico a mano.
  - Es difícil probar que el código del analizador morfológico es correcto desde el punto de vista formal y que realmente comprende todos los casos posibles.
- El enfoque de esta sección corrige estos problemas, mediante el uso de herramientas formales como las expresiones regulares para representar las unidades sintácticas.

Analizador morfológico implementación

Alternativas para la construcción del analizador morfológico (II)

- En esta sección se verá:
  - Diseño de un autómata finito y uso de un simulador de autómatas finitos para su ejecución:
    - Desde las expresiones regulares del lenguaje regular considerado.
    - Desde la gramática regular obtenida a partir de la gramática independiente del contexto.
    - Diseño directo del autómata finito (únicamente para ejemplos muy sencillos).



Uso de herramientas formales (II)

Expresiones regulares de las unidades sintácticas

- Por ejemplo para la gramática que se vio de ASPLE, las expresiones regulares para cada una de las unidades sintácticas sería:

```
palabra reservada={begin, ;, end, bool, int, ref, ,
                   call, :=, if, then, fi, else,
                   while, do, repeat, until, input,
                   output, +, -, *, (, ), =, <=, >,
                  case, esac, :, procedure}

bool constant={true, false}

int constant ={0,···,9}+

id ={a,···, z, A,···, Z}{a,···, z, A,···, Z, 0,···, 9}*
```

Uso de herramientas formales (III)

Expresiones regulares de las unidades sintácticas

- A partir de las expresiones regulares anteriores, se puede obtener una completa para todas las unidades sintácticas haciendo su unión:

```
Unidad Sintáctica={ begin, ;, end, bool, int, ref, ,,
                    call, :=, if, then, fi, else,
                    while, do, repeat, until, input,
                    output, +, -, *, (, ), =, <=, >,
                    case, esac, :, procedure}
                ∪
                    { true, false}
                ∪
                    {0,···,9}+
                ∪
                    {a,···, z, A,···, Z}{a,···, z, A,···, Z,
0,···, 9}*
```


Uso de herramientas formales (IV)

Gramática regular de las unidades sintácticas

- También se puede comprobar que hay una gramática regular para cada una de esas unidades sintácticas:

```
<palabra reservada> ::= begin | ; | end | bool | int | ref | ,
                      |   call | := | if | then | fi | else
                      |   while | do | repeat | until | input
                      |   output | + | - | * | ( | ) | = | <= | >
                      |   case | esac | : | procedure
<bool constant> ::= true | false
<int constant> ::= 0 | ··· | 9 | 0 <int constant> | ···
                 |   9 <int constant>
<id> ::= A | ··· | Z | a | ··· | z
     |   A <rest id> | ··· | Z <rest id>
     |   a <rest id> | ··· | z <rest id>
<rest id> ::= A | ··· | Z | a | ··· | z | 0 | ··· | 9
           |   0 <rest id> | ··· | 9 <rest id>
           |   A <rest id> | ··· | Z <rest id>
           |   a <rest id> | ··· | z <rest id>
```

Uso de herramientas formales (V)

Gramática regular de las unidades sintácticas

- A partir de la gramática anterior es fácil obtener una gramática regular completa para todas las unidades sintácticas.

- Para ello se debe:
  - Diseñar un nuevo símbolo para la unidad sintáctica (**<US>**, por ejemplo)
  - Añadir reglas para que tengan
    - Como parte izquierda el símbolo nuevo.
    - Todas las partes derechas,
  - Mantener las reglas de los símbolos no terminarles que aparecen en la parte derecha de cada regla.
  - Eliminar las reglas superfluas.

Uso de herramientas formales (VI)

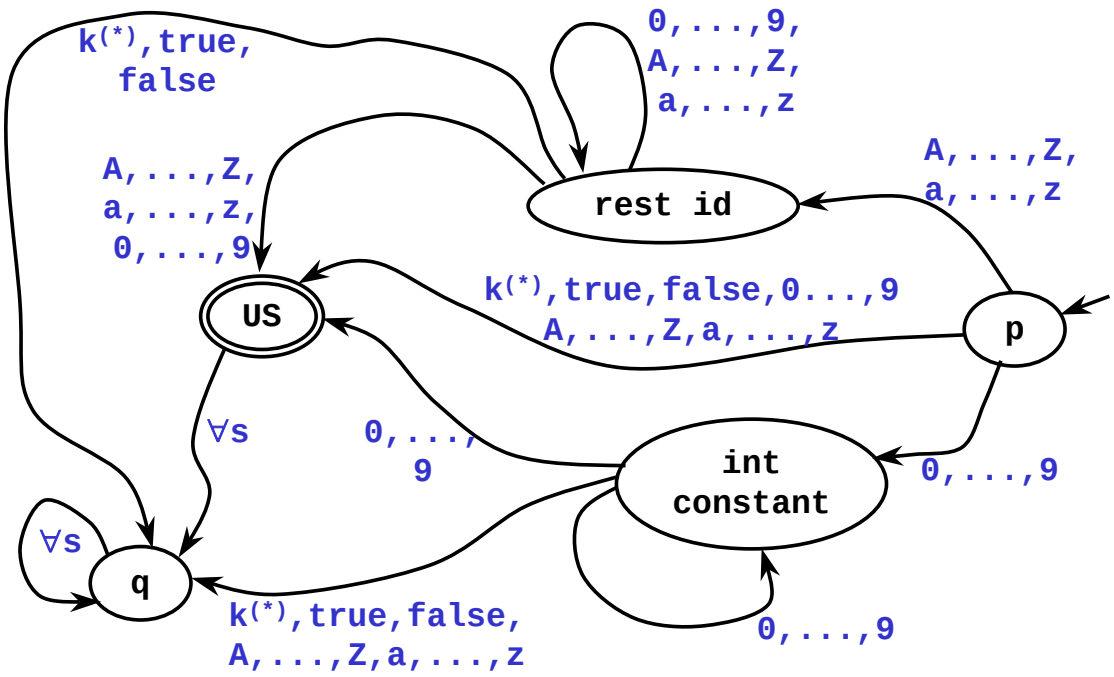
Gramática regular de las unidades sintácticas

```
<US> ::= begin | ; | end | bool | int | ref | ,
       |   call | := | if | then | fi | else
       |   while | do | repeat | until | input
       |   output | + | - | * | ( | ) | = | <= | >
       |   case | esac | : | procedure
       |   true | false
       |   0 | ··· | 9 | 0 <int constant> | ···
       |   9 <int constant>
       |   A | ··· | Z | a | ··· | z
       |   A <rest id> | ··· | Z <rest id>
       |   a <rest id> | ··· | z <rest id>

<int constant> ::= 0 | ··· | 9 | 0 <int constant> | ···
                 |   9 <int constant>
<rest id> ::= A | ··· | Z | a | ··· | z | 0 | ··· | 9
            |   0 <rest id> | ··· | 9 <rest id>
            |   A <rest id> | ··· | Z <rest id>
            |   a <rest id> | ··· | z <rest id>
```

Uso de herramientas formales (VII)

Autómata asociado a la gramática



Uso de herramientas formales (VII)

Otros patrones

- ASPLE proporciona al programador un subconjunto de los tipos que los lenguajes de programación de alto nivel suelen proporcionar.
- Otros tipos de datos frecuentes en los lenguajes de programación:
  - Números reales, como `3.45`, `.44`, `-5.`, `3.45E2`, `.44E-2`, `-5.E123`

```
<real>::=<punto fijo>
      |  <punto fijo><exponente>
<entero>::=<int constant>
        |-<int constant>
<punto fijo>::=<entero>.<int constant>
            |  .<int constant>
            |  <entero>.
<exponente>::=E<entero>
```

  - Cadenas y caracteres, como “”, “hola amigos”, ‘a’,...,‘z’

```
<literal>::=“” | “<cadena>(*)”
<carácter>::=‘<simbolo>(*)’
```

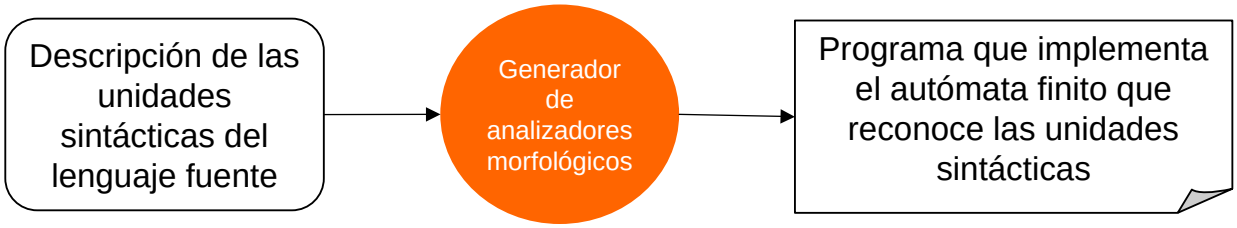
5.3.3

Uso de herramientas de generación de analizadores (Flex)

Uso de herramientas de generación de analizadores (I)

Generadores de analizadores morfológicos

- Adicionalmente, se dispone de herramientas que automatizan la codificación de estos programas: los generadores de analizadores morfológicos.
- En general, estas herramientas reciben como entrada la descripción de las unidades sintácticas del lenguaje fuente (en un formato determinado), y generan un programa que se corresponde con el autómata que reconoce las unidades sintácticas especificadas.



- En el compilador que se desarrollará en las prácticas del curso, se utilizará para la construcción del analizador morfológico el generador Lex/Flex.

Uso de herramientas de generación de analizadores (II)

FLEX

- Flex es capaz de generar un analizador morfológico en C que va procesando char a char el texto de entrada, buscando cadenas que se correspondan con las expresiones regulares que se le indiquen para las unidades sintácticas.
- Si encuentra varias expresiones con las que una entrada coincide, la norma es elegir la cadena más larga.
  - ⇨ Por ej: “23.56” se reconoce como float (no como integer).

```
{DIGIT}+    { printf( “ (integer, ‘%s’) ", yytext); }

{DIGIT}+"."{DIGIT}* { printf( “(float, ‘%s’)", yytext);}
```

- Una vez que encuentra la expresión regular con la que coincide la cadena:
  - La etiqueta correspondiente a esa expresión regular se guarda en yytext.
  - Se ejecuta la acción asociada a esa etiqueta (si la hubiera).
- Después de reconocer una unidad sintáctica, se comienza a buscar nuevas unidades sintácticas a partir de ella

Uso de herramientas de generación de analizadores (III)

Estructura de un programa FLEX

```
%{
   Includes y código previo
%}

Definiciones Flex

%%

Reglas (Expr. Regulares)

%%

Código de usuario
```

Uso de herramientas de generación de analizadores (IV)

Ejemplo de reglas FLEX para un analizador morfológico sencillo

```
DIGIT                [0-9]
ID                   [a-z][a-z0-9]*

{DIGIT}+             { printf( “ (integer, ‘%s’) ", yytext); }

{DIGIT}+"."{DIGIT}* { printf( “(float, ‘%s’)", yytext);}

if|then|begin|end|procedure|function
                     { printf( “(%s,: ‘%s’)", yytext, yytext);}

{ID}                 printf( “(id, ‘%s’)", yytext );

"+"|"-"|"*"|"/"      printf( “(mathop, ‘%s’)", yytext );

[ \t\n]+             /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );
```


5.4

Acciones semánticas

Acciones semánticas (I)

Otras tareas del analizador morfológico

- El compilador puede delegar también tareas semánticas al analizador morfológico.

- Algunas de estas tareas pueden ser las siguientes:
  - Almacenar la información asociada a un identificador en la tabla de símbolos del identificador.
  - Cálculo de los valores numéricos de las constantes.

- Estas tareas varían en función de
  - Los objetivos de los traductores / intérpretes.
  - La división de tareas que se decida en la implantación del traductor / intérprete.

Analizador morfológico

Bibliografía

[Alf06] Alfonseca, M., de la Cruz, M., Ortega, A., Pulido, E. "*Compiladores e intérpretes: teoría y práctica*", Pearson, 2006.

[Alf] "*Teoría de Autómatas y lenguajes formales*" M. Alfonseca, Roberto Moriyón, Enrique Alfonseca.

[Hop] *"Introducción a la teoría de autómatas, lenguajes y computación*" Hopcroft, J.; Motwani, R.; Ullman, J.