Introducción a la tecnología de compiladores

4

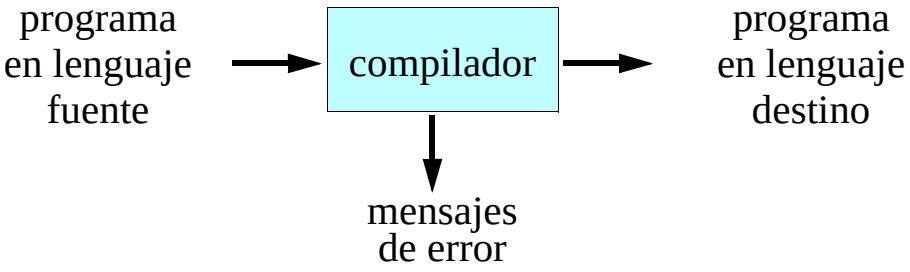
Notas:

UNIVERSIDAD DE CANTABRIA

1. Compiladores

UNIVERSIDAD DE CANTABRIA

“Un compilador es un programa que lee un programa escrito en un lenguaje, y lo traduce a un programa equivalente en otro lenguaje.”



Durante la traducción el compilador informa de la presencia de errores en el programa fuente.

Notas:

UNIVERSIDAD DE CANTABRIA

Existen multitud de compiladores para un gran número de lenguajes fuente y lenguajes destino.

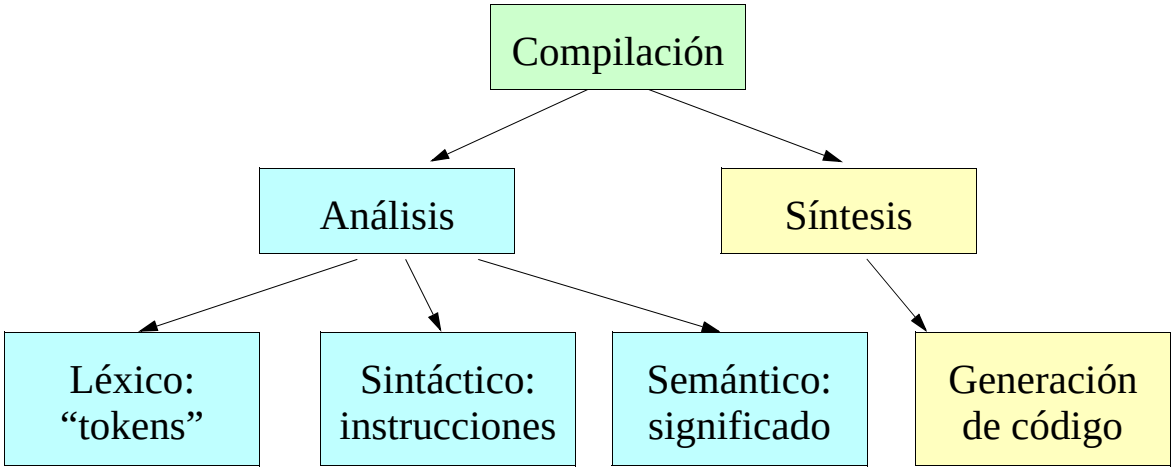
Los compiladores se utilizan tanto para lenguajes de programación tradicionales, como Fortran, C o Ada, o para aplicaciones especializadas como por ejemplo lenguajes de descripción de hardware, lenguajes de programación de robots, etc.

Los primeros compiladores aparecieron a primeros de los años 50, como resultado de proyectos para la traducción de fórmulas aritméticas en código máquina.

Los primeros compiladores eran costosos de implementar. Hoy en día existen técnicas sistemáticas para construir compiladores que hacen más sencillo el proceso.

Análisis y síntesis

UNIVERSIDAD DE CANTABRIA



Notas:

UNIVERSIDAD DE CANTABRIA

Existen dos partes importantes en la compilación:

- *Etapa de análisis*: Parte el programa fuente en sus piezas constituyentes y crea una representación intermedia del mismo.
  - Análisis léxico: separación de cada elemento componente del programa ("token")
  - Análisis sintáctico: separación de cada instrucción o sentencia del lenguaje, que agrupa varios componentes léxicos o "tokens".
  - Análisis semántico: Se revisa el programa fuente para comprobar que las reglas semánticas del lenguaje (aquellas relativas al significado de las distintas instrucciones) se cumplen. Un ejemplo de regla semántica es la comprobación de tipos en las expresiones.
- *Etapa de síntesis*: Construye el programa destino deseado a partir de una descripción en un lenguaje de representación intermedia.

De las dos partes de la compilación, la síntesis es la que requiere las técnicas más especializadas, aunque en los lenguajes de programación modernos (Ada, C++, Java) la parte de análisis está alcanzando una gran complejidad.

Durante la fase de análisis la estructura del programa se guarda en una estructura de datos especial que suele ser un árbol: el árbol sintáctico.

Análisis y síntesis (cont.)

UNIVERSIDAD DE CANTABRIA

Algunas herramientas presentan también una etapa de análisis:

- **Editores orientados al lenguaje**
- **Impresión con formato (“pretty printer”)**
- **Comprobadores estáticos de programas**
- **Intérpretes**

Notas:

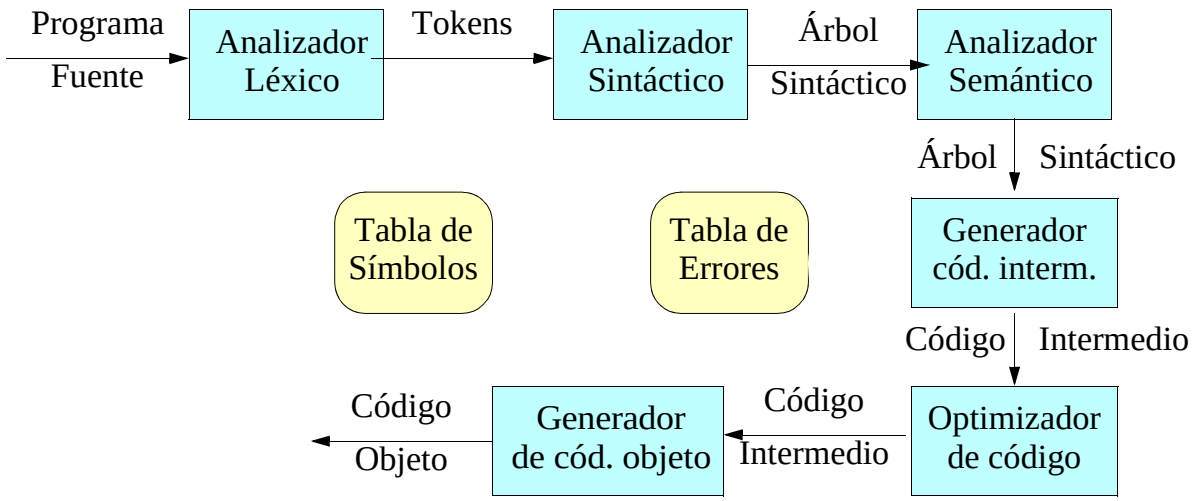
UNIVERSIDAD DE CANTABRIA

La etapa de análisis se encuentra en muchas herramientas además de los compiladores:

- Editores orientados al lenguaje: Además de facilitar la introducción de texto por el teclado analiza el programa fuente y proporciona la estructura y jerarquía necesarias. Por ejemplo, puede comprobar que la sintaxis es correcta, proporcionar palabras clave, etc.
- Impresión con formato (“Pretty Printer”): Analiza el programa fuente y lo imprime de forma que la estructura del programa aparece claramente visible.
- Comprobadores estáticos: Permiten analizar un programa y descubrir errores potenciales sin necesidad de ejecutar el programa. Por ejemplo, se pueden detectar zonas de código que no se ejecutarán nunca, variables no inicializadas, errores sintácticos, comprobación de tipos, etc.
- Intérpretes: En lugar de producir un programa destino mediante un proceso de traducción, el intérprete ejecuta las operaciones que especifica el programa fuente.

Fases de un compilador

UNIVERSIDAD DE CANTABRIA



Notas:

UNIVERSIDAD DE CANTABRIA

En cada fase de un compilador se transforma el programa fuente de una representación a otra. Las tres primeras fases forman la etapa de análisis, mientras las tres últimas forman la etapa de síntesis.

La tabla de símbolos es una estructura de datos que almacena los identificadores utilizados en el programa fuente así como los atributos de cada identificador. Estos atributos pueden proporcionar información sobre el tipo del identificador, su tamaño, su rango de visibilidad, sus argumentos (en caso de procedimientos), etc.

La tabla de símbolos tiene operaciones para encontrar un identificador rápidamente, y leer sus atributos o modificarlos. Asimismo, permite introducir nuevos identificadores. Cada una de las fases de compilación puede realizar modificaciones de los registros de una tabla de símbolos, generalmente añadiendo más atributos a medida que se van conociendo.

El manejador de errores es un módulo que gestiona las acciones a realizar por cada uno de los errores encontrados en las diferentes fases de la compilación. En general, es deseable que el manejador de errores permita la continuación del proceso de compilación, con objeto de permitir encontrar más errores en el programa. Las fases de análisis sintáctico y semántico son habitualmente las que más errores encuentran.

Ejemplo de análisis y síntesis

UNIVERSIDAD DE CANTABRIA

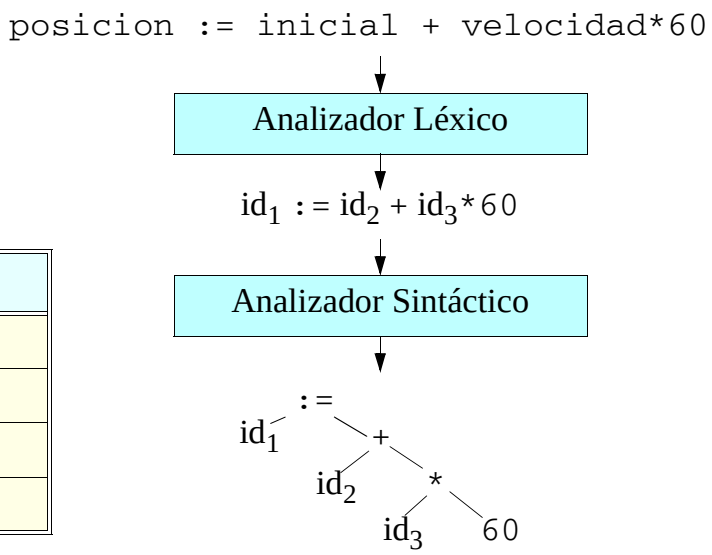
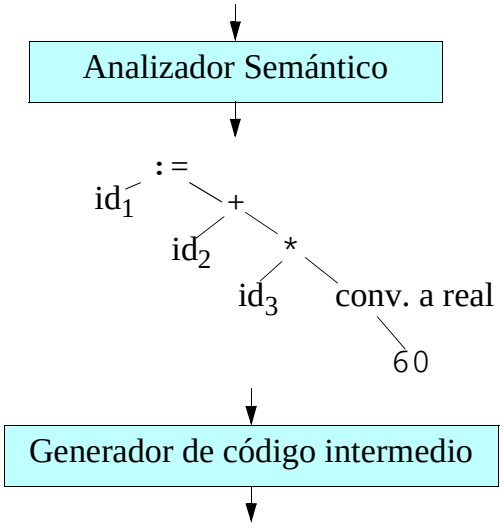


Tabla de símbolos		
1	posicion	...
2	inicial	...
3	velocidad	...
4	...	...

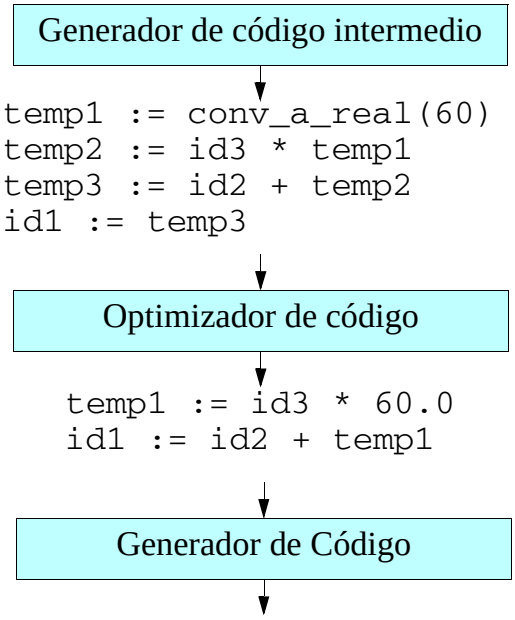
Notas:

UNIVERSIDAD DE CANTABRIA



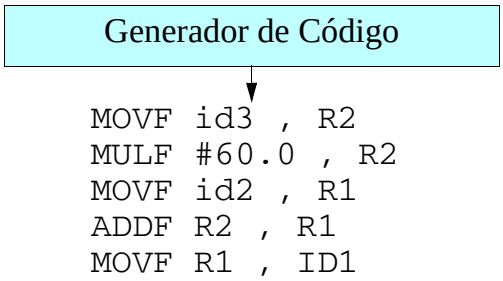
Ejemplo (cont.)

UNIVERSIDAD DE CANTABRIA



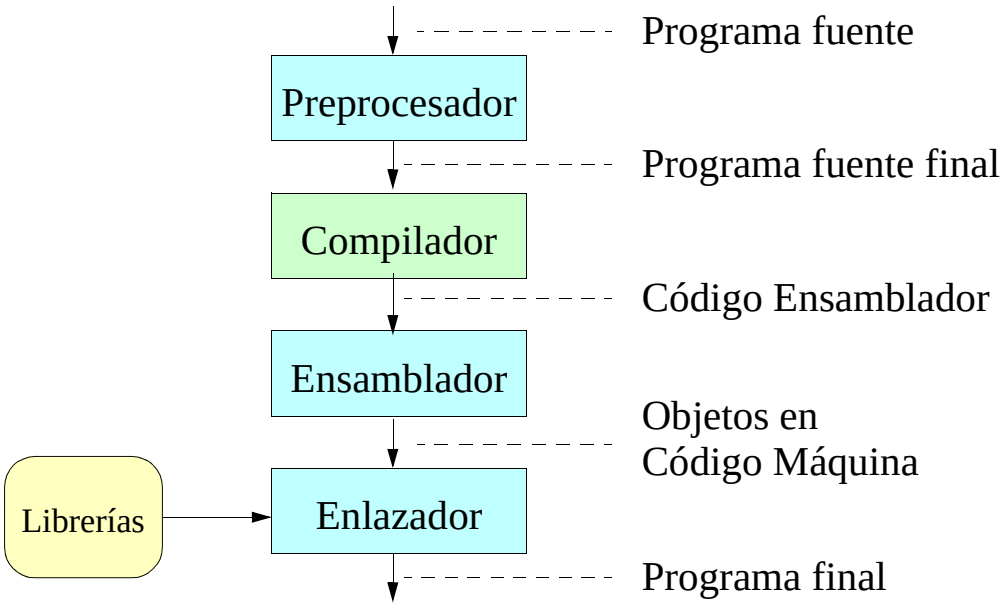
Notas:

UNIVERSIDAD DE CANTABRIA



El contexto de un compilador

UNIVERSIDAD DE CANTABRIA



Notas:

UNIVERSIDAD DE CANTABRIA

Además de un compilador, otros programas pueden ser necesarios para crear un programa destino ejecutable:

- Preprocesador: Un programa fuente puede estar dividido en módulos almacenados en ficheros diferentes. La tarea de recopilar el código fuente almacenado en estos ficheros puede ser encomendada a un preprocesador. Asimismo, un preprocesador puede expandir las macros convirtiéndolas en instrucciones ejecutables.
- Ensamblador: Muchos compiladores proporcionan el programa final en lenguaje ensamblador. Para poder obtener un programa ejecutable es preciso ensamblar este programa final con un ensamblador convencional.
- Enlazador.Esta herramienta toma código máquina relocalizable de los diferentes objetos compilados y de librería, modifica las direcciones relocalizables para situarlas a los valores absolutos adecuados, y crea el programa ejecutable.

2. Análisis léxico

UNIVERSIDAD DE CANTABRIA

Convierte una cadena de caracteres que conforma el programa fuente en un grupo de "palabras", que son secuencias de caracteres con significado propio. Ejemplo:

```
if Existe then
posicion:=60;
end if;
```

1. la palabra reservada "`if`"
2. la expresión booleana "`Existe`"
3. la palabra reservada "`then`"
4. el identificador "`posicion`"
5. el símbolo de asignación "`:=`"
6. la constante "`60`"
7. el final de instrucción "`;`"
8. la palabra reservada "`end`"
9. la palabra reservada "`if`"
10. el final de instrucción "`;`"

Notas:

UNIVERSIDAD DE CANTABRIA

Durante el análisis léxico el programa fuente, que está constituido por una cadena de caracteres, se lee de principio a fin y se descompone en un conjunto de palabras, cada una de ellas con significado propio.

Estas palabras son identificadas y clasificadas en diferentes grupos. Por ejemplo, las palabras de un lenguaje de programación pueden ser identificadores, constantes, operadores, palabras reservadas del lenguaje, etc.

Los espacios en blanco que separan los caracteres que forman las palabras son normalmente eliminados durante la fase de análisis léxico. También lo son otros separadores como tabuladores, saltos de línea, etc. También se pueden eliminar en esta fase los comentarios.

Los errores detectados en la fase de análisis léxico suelen ser escasos, porque en esta fase se dispone de muy poca información sobre el programa. Un error que sí se puede detectar es la lectura de un carácter que no corresponde al alfabeto permitido para el lenguaje.

Reconocimiento de lexemas y la tabla de símbolos

UNIVERSIDAD DE CANTABRIA

Para distinguir unas palabras (o lexemas) de otras se utilizan *patrones* de reconocimiento.

En muchos casos los patrones se describen mediante expresiones regulares.

Los identificadores pueden también sustituirse por referencias a la tabla de símbolos, para una utilización más eficiente.

La tabla de símbolos asocia a cada identificador un número, así como una serie de atributos (tipo de datos, etc.).

Notas:

UNIVERSIDAD DE CANTABRIA

Para distinguir unas palabras de otras durante el análisis léxico, se utilizan determinados patrones de reconocimiento. Por ejemplo, en una expresión aritmética, los símbolos de operadores, como “+”, “-”, “*”, “/”, son reconocidos como lexemas, y separan al resto de los lexemas, que a su vez pueden ser palabras (nombres de variables) o números.

Los patrones de reconocimiento se suelen describir utilizando diversas notaciones. La más popular son las expresiones regulares, que veremos a continuación.

En ocasiones, cuando el analizador léxico encuentra un identificador lo introduce en la tabla de símbolos. En este caso, la representación del programa que produce el analizador léxico presenta referencias a los identificadores almacenados en la tabla de símbolos.

Expresiones regulares

UNIVERSIDAD DE CANTABRIA

Permiten describir conjuntos de strings. El uso más habitual es para comprobar si un string dado se corresponde con un patrón, descrito mediante una expresión regular

Notación general para REs:

Símbolo	Descripción
`|`	Or (alternativa)
`()`	Agrupar una subexpresión
`*`	0 ó más veces
`?`	0 ó 1 vez
`+`	1 o más veces
`{n,m}`	entre n y m veces

Notas:

UNIVERSIDAD DE CANTABRIA

Las expresiones regulares permiten definir conjuntos de strings. Con ellos, podemos comprobar si una palabra cumple un determinado patrón de reconocimiento.

Por ejemplo, un identificador Ada es una palabra que empieza con una letra, y luego tiene letras, cifras numéricas, o el carácter subrayado. Esto se expresa así:

- identificador=letra(letra | dígito | _)*

Donde:

- las letras corresponden al conjunto [a-zA-Z]
- los dígitos al conjunto [0-9]
- y el carácter subrayado se expresa como él mismo.

Reglas para describir expresiones regulares

UNIVERSIDAD DE CANTABRIA

Descripción	Sintaxis
expresar un carácter especial x	`\x o "x"`
a seguido de b	`a/b`
carácter a al final de una línea	`a$`
carácter a al principio de una línea	`^a`
caracteres a, b ó c	`[abc]`
cualquier carácter excepto a, b ó c	`[^abc]`
caracteres entre A y E	`[A-E]`
tabulador - nueva línea	`\t - \n`
cualquier carácter excepto \n	`.`
definición regular	`nombre exp_regular`

Notas:

UNIVERSIDAD DE CANTABRIA

Ejemplos de expresiones regulares
`signo_opcnal(+ | -)?`
`digito[0-9]`
`digitos{digito}+`
`mantisa_real{signo_opcnal} {digitos} . {digitos}`
`exp_opcnal(E {signo_opcnal} {digitos})?`
`numero_real{mantisa_real} {exp_opcnal}`

El constructor de analizadores léxicos `lex`

UNIVERSIDAD DE CANTABRIA

Lex es un constructor de analizadores léxicos.

Opera mediante una descripción de la gramática hecha mediante expresiones regulares.

Genera un programa en lenguaje C que sirve como analizador léxico.

Existe un equivalente en lenguaje Ada: aflex

`http://www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html`

Notas:

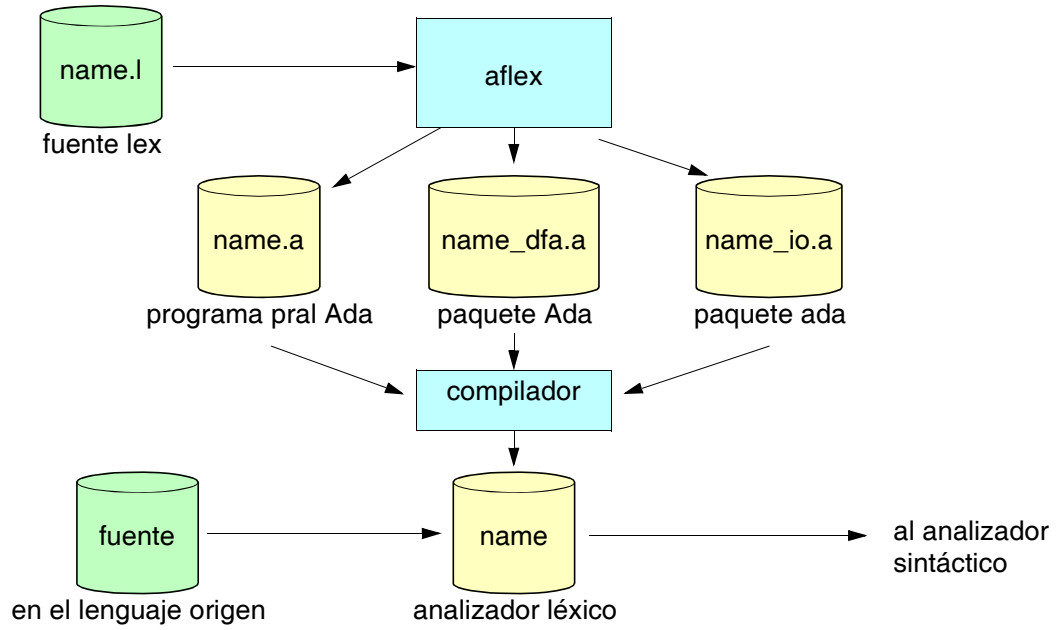
UNIVERSIDAD DE CANTABRIA

Lex es una herramienta que se ha utilizado para especificar analizadores léxicos para una variedad de lenguajes. Normalmente se le llama compilador Lex a la herramienta, y lenguaje Lex a sus especificaciones de entrada.

Existe un equivalente a “lex” en Ada, denominado “aflex”. La herramienta lex genera un programa C que sirve como analizador léxico; aflex genera también un analizador léxico, pero en lenguaje Ada.

Uso de la herramienta `aflex`

UNIVERSIDAD DE CANTABRIA



Notas:

UNIVERSIDAD DE CANTABRIA

Generalmente, lex o aflex se utilizan en la forma indicada en la figura de arriba. Primero se prepara una especificación de la gramática en lenguaje Lex. Posteriormente, se utiliza la herramienta para producir el código fuente del programa analizador léxico. Este código fuente debe ser compilado, lo que produce el analizador léxico. Con el analizador léxico resultante, se puede realizar el análisis léxico de un código fuente tantas veces como sea necesario.

La herramienta aflex crea ficheros Ada terminados en ".a", que por tanto no se adhieren a las convenciones de nombres del compilador gnat. Además, crea los paquetes en un solo fichero, con la especificación y el cuerpo en el mismo fichero. Esto también es contrario a las reglas del gnat.

Para estas situaciones, existe una herramienta llamada "gnatchop", que permite crear los ficheros con los nombres apropiados, a partir de otros ficheros. En el caso del ejemplo de la figura, habrá que ejecutar:

```
gnatchop -w name.a
gnatchop -w name_dfa.a
gnatchop -w name_io.a
```

Donde "name" es el nombre de la especificación lex original. Luego hay que construir el ejecutable:

```
gnatmake name
```

Especificaciones en `lex`

UNIVERSIDAD DE CANTABRIA

Constan de tres partes:

```
definiciones
%%
reglas de traducción
%%
código auxiliar
```

Las *definiciones* son definiciones regulares que toman la forma:

```
nombre expresión_regular
```

Las *reglas de traducción* se describen de la forma:

```
patrón {acción}
```

Patrón es una expresión regular; *Acción* es un trozo de código

Notas:

UNIVERSIDAD DE CANTABRIA

Las especificaciones en Lex constan de las tres partes descritas arriba: definiciones, reglas de traducción, y código auxiliar en C (o Ada en el caso de aflex).

Las definiciones son definiciones regulares. Por ejemplo:

```
letra          [a-zA-Z]
digito         [0-9]
identificador  {letra}({letra}|{digito}|_)*
```

Las reglas de traducción describen patrones y acciones a realizar en caso de que en el texto de entrada del analizador léxico se encuentre ese patrón. Las acciones son fragmentos de código C, en el caso de lex, o Ada, en el caso de aflex. Por ejemplo:

```
{identificador} {Put_Line ("Encontrado un identificador");}
{digito}+       {Put_Line ("Encontrado un número");}
```

En los nombres de las definiciones se distinguen mayúsculas de minúsculas. Por ejemplo, no es lo mismo {digito} que {Digito}.

Estructura del código auxiliar

UNIVERSIDAD DE CANTABRIA

El código auxiliar se escribe con la estructura

```
sección inicial de código
##
sección final de código
```

El código Ada generado tendrá la estructura siguiente:

```
with Name_Dfa, Name_IO, Text_IO;
sección inicial de código (debe incluir el tipo Token)

function YYLex return Token is
begin
   -- definida por la herramienta lex
end;

sección final de código
```


Notas:

UNIVERSIDAD DE CANTABRIA

Un ejemplo de código auxiliar:

```
procedure Name is
   type Token is (End_Of_Input, Error);
   Tok : Token;

##

begin
   while Tok /= End_of_Input loop
      Tok:=YYLex;
   end loop;
end Name;
```

La función YYLex se crea automáticamente, y se coloca en el lugar donde se han escrito los caracteres ##. Para que esta función sea válida, es imprescindible que en la sección inicial de código se haya creado el tipo enumerado Token, con al menos los dos valores End_Of_Input y Error.

Aspectos a tener en cuenta en la especificación `lex`

UNIVERSIDAD DE CANTABRIA

El texto que verifica el patrón de una regla de traducción se puede obtener con la función predefinida `YYText`.

Es habitual que el código de una regla de traducción acabe con una instrucción:

```
return valor;
```

donde `valor` es un valor del tipo enumerado `Token`, creado en la sección inicial de código auxiliar. Esto es especialmente habitual si se enlaza `lex` con `yacc` (analizador sintáctico)

Cuando un string puede cumplir dos reglas de traducción, se elige:

1. **El string más largo**
2. **Si son iguales, la regla que aparece en primer lugar**

Notas:

UNIVERSIDAD DE CANTABRIA

Un aspecto de importancia al escribir las reglas de traducción es cómo se resuelven las ambigüedades. Por ejemplo, si tenemos las definiciones

```
lazo          "lazo"
lazo_while    "lazo_while"
```

el string “lazo_while” cumpliría las dos definiciones.

En lex las ambigüedades se resuelven usando siempre el string más largo de los que cumplen varias reglas. Si hay varias reglas que son cumplidas por strings de la misma longitud, entonces se elige la regla colocada en primer lugar.

Ejemplo con `lex`

UNIVERSIDAD DE CANTABRIA

El siguiente ejemplo convierte un texto con insultos en otro con palabras más "correctas" :-)

Especificación Lex, en el fichero `bien_educado.l`:

```
-- definiciones
espacio             " "|\n
palabra             [a-zA-Z]+{espacio}
tonto               ("tonto"{espacio})|("ignorante"{espacio})
tonta               "tonta"{espacio}
tonto_de            "tonto"{espacio}"de"l?{espacio}{palabra}
idiota              "idiota"{espacio}
nueva_linea         \n

%%
```


Notas:

UNIVERSIDAD DE CANTABRIA

```
   -- reglas de traducción
{tonto}             {Put("listo ");}
{tonta}             {Put("lista ");}
{idiota}            {Put("distinguido ");}
{tonto_de}          {Put("amable caballero ");}
{nueva_linea}       {New_Line;}
.                   {Put(YYText);}

%%

   -- código auxiliar inicial
with Ada.Text_IO; Use Ada.Text_IO;
procedure Bien_Educado is
type Token is (end_of_input,error);
tok : token;
```

Ejemplo (cont.)

UNIVERSIDAD DE CANTABRIA

```
##

-- código auxiliar final
begin
     loop
          tok:=YYLex;
          exit when Tok = end_of_input;
     end loop;
end Bien_Educado;
```


Notas:

UNIVERSIDAD DE CANTABRIA

El ejemplo se ha utilizado para procesar el siguiente texto:

```
Juan le dijo al ignorante de Pedro que no hiciera
caso del tonto de baba de su hermano,
ya que el idiota de el estaba
llamando tonta a su hermana.
```

El resultado obtenido es:

```
Juan le dijo al listo de Pedro que no hiciera
caso del amable caballero de su hermano,
ya que el distinguido de el estaba
llamando lista a su hermana.
```

3. Análisis sintáctico

UNIVERSIDAD DE CANTABRIA

Agrupa las palabras del programa fuente en "frases" anidadas jerárquicamente con un significado común.

Normalmente utiliza unas reglas sintácticas para describir la gramática del lenguaje fuente.

Se describen habitualmente mediante gramáticas sin contexto, también llamadas notación BNF (Backus-Naur Form)

Para algunos tipos de gramáticas es posible construir el analizador con herramientas automáticas

Notas:

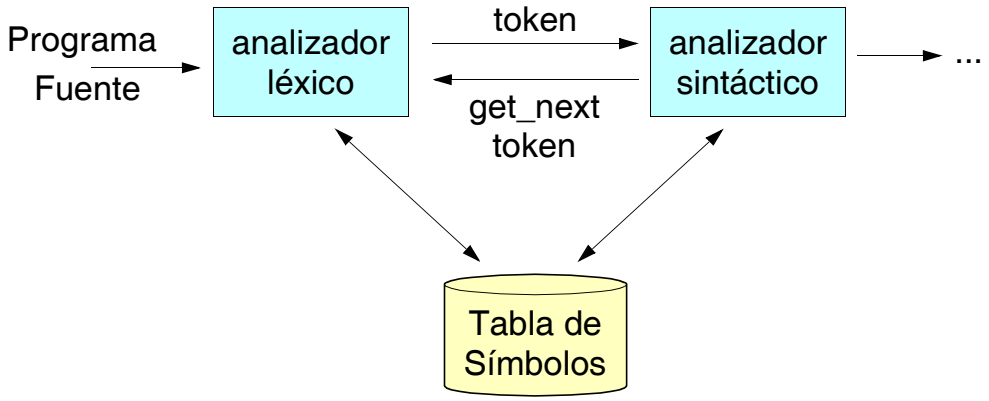
UNIVERSIDAD DE CANTABRIA

Todos los lenguajes de programación tienen un conjunto de reglas que describen la estructura sintáctica de los programas correctos. La sintaxis de las construcciones de los lenguajes de programación pueden describirse mediante gramáticas sin contexto (context-free grammars), también llamada notación BNF (Backus-Naur Form). Las gramáticas ofrecen significativas ventajas tanto para los diseñadores del lenguaje como para los escritores de compiladores:

- Una gramática suministra una forma precisa y fácil de entender de la especificación sintáctica de un lenguaje de programación.
- Para ciertas clases de gramáticas, se pueden construir analizadores sintácticos con herramientas automáticas. Además, la construcción del analizador puede revelar ambigüedades gramaticales no deseadas.
- Una gramática adecuadamente diseñada aporta al lenguaje de programación una estructura adecuada tanto para su traducción a código objeto, como para la detección de errores.
- Como los lenguajes tienen una vida larga y pueden evolucionar añadiéndoseles nuevas construcciones, éstas pueden ser añadidas más fácilmente cuando la implementación existente está basada en una descripción gramatical formal.

Interacción entre analizador léxico y sintáctico

UNIVERSIDAD DE CANTABRIA



Notas:

UNIVERSIDAD DE CANTABRIA

Como se ha señalado, el analizador léxico es la primera fase de un compilador. Su principal tarea es leer los caracteres de entrada y producir como salida una secuencia de “tokens” o lexemas, que el analizador sintáctico o “parser” utilizará para el análisis sintáctico.

La interacción entre el analizador léxico y el analizador sintáctico se muestra en la figura de arriba. Normalmente el analizador léxico se escribe como un subprograma del analizador sintáctico, al que éste llama para obtener cada “token”.

En el caso de las herramientas lex y yacc, el analizador léxico suele ser una función llamada YYLex, que retorna cada vez un dato del tipo Token diferente.

Ambos analizadores pueden compartir una tabla de símbolos, que sirve para traducir identificadores a números, más fáciles de manejar.

Gramáticas sin contexto

UNIVERSIDAD DE CANTABRIA

Una gramática describe la estructura jerárquica de un lenguaje. Por ejemplo, la instrucción "if"

```
if expresion_logica then instrucciones end if;
```

se puede expresar en notación BNF con la siguiente regla:

```
instruccion_if -> if expresion then instrucciones end if;
```

La gramática tiene cuatro componentes:

1. **conjunto de lexemas o tokens (if, then, end)**
2. **conjunto de no-terminales (expresion, instrucciones)**
3. **conjunto de reglas, cada una describiendo un no-terminal**
4. **la designación de un no-terminal como el comienzo**

Notas:

UNIVERSIDAD DE CANTABRIA

Una gramática describe la estructura jerárquica de las construcciones de un lenguaje de programación.

Una gramática sin contexto tiene los siguientes cuatro componentes:

- Un conjunto de lexemas o "tokens", también denominados símbolos terminales.
- Un conjunto de elementos no terminales, que se asocian a strings, y cuyo significado se describe en alguna de las reglas de la gramática
- Un conjunto de reglas de "producción", cada una de ellas constituida por un no-terminal, una flecha (o símbolo equivalente) y una secuencia de tokens y no-terminales.
- La designación de uno de los no-terminales como el símbolo de comienzo. Normalmente se sigue el criterio de poner en primer lugar la regla que define el no-terminal de comienzo.

Por ejemplo, la siguiente gramática define una expresión binaria con números enteros:

```
exp_simple -> exp_simple + digito
exp_simple -> exp_simple - digito
exp_simple -> digito
digito -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Herramienta yacc

UNIVERSIDAD DE CANTABRIA

Sirve para generar analizadores sintácticos, utilizando una descripción gramatical expresada con BNF

La herramienta yacc combina en la gramática instrucciones en lenguaje C para la construcción del analizador sintáctico

Existe una herramienta similar para lenguaje Ada: ayacc

Una descripción ayacc tiene tres partes:

```
declaraciones
%%
reglas de traducción
%%
código auxiliar Ada
```


Notas:

UNIVERSIDAD DE CANTABRIA

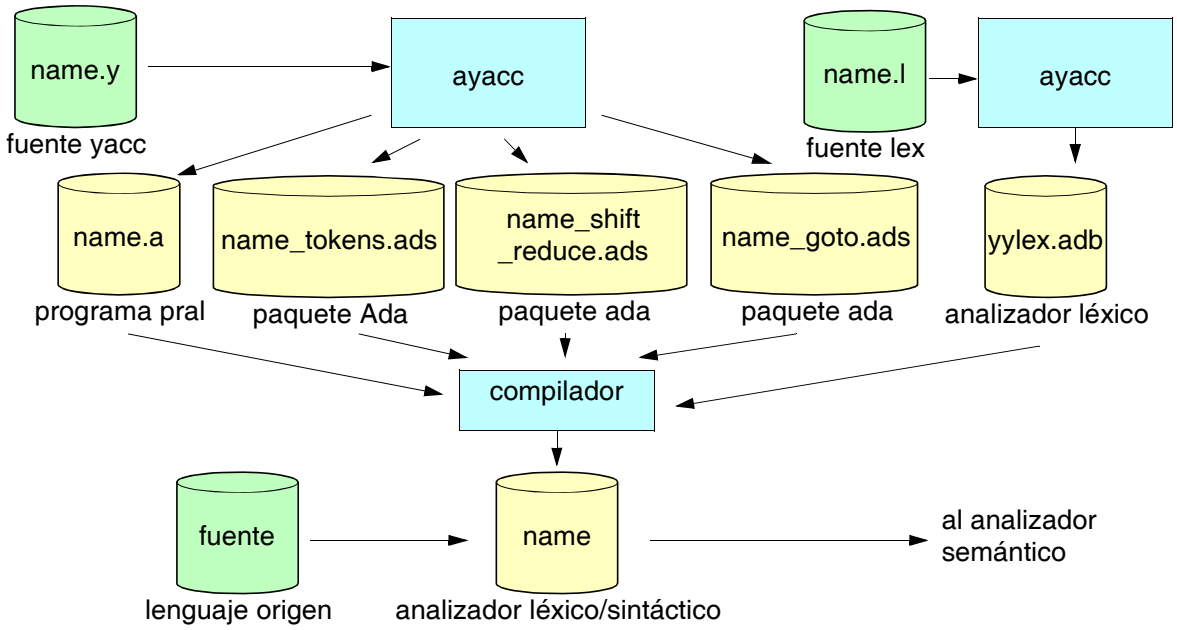
Yacc es una herramienta para la generación de analizadores sintácticos o “parsers”. Fue desarrollada en los años 70 por S.C. Johnson, y está disponible junto a la herramienta lex como una utilidad estándar en los sistemas UNIX. Se ha utilizado para generar cientos de compiladores.

La herramienta Yacc utiliza una descripción de la gramática mediante la notación BNF. Para cada regla gramatical, se pueden especificar acciones a realizar por el analizador sintáctico, expresadas en lenguaje C. Al final, la herramienta genera un programa C, que después de compilado y enlazado con un analizador léxico (posiblemente producido con lex), produce el programa analizador sintáctico.

Existe una herramienta similar denominada ayacc para lenguaje Ada. En esa herramienta las acciones a realizar se expresan en lenguaje Ada, y el programa resultante es un programa Ada.

Uso de la herramienta ayacc

UNIVERSIDAD DE CANTABRIA



Notas:

UNIVERSIDAD DE CANTABRIA

Generalmente, ayacc se utiliza combinado con la herramienta de generación de analizadores léxicos lex, en la forma indicada en la figura de arriba. Primero se prepara una especificación de la gramática en lenguaje ayacc, así como el analizador léxico en lex. Posteriormente, se utiliza la herramienta yacc para producir el código fuente del programa analizador sintáctico. Este código fuente debe ser compilado, lo que produce el analizador sintáctico. Con el analizador sintáctico resultante, que utiliza internamente el analizador léxico para leer el texto de entrada, se puede realizar el análisis léxico/sintáctico de un código fuente tantas veces como sea necesario.

La herramienta ayacc crea ficheros Ada terminados en ".a", que por tanto no se adhieren a las convenciones de nombres del compilador gnat. Además, crea los paquetes en un solo fichero, con la especificación y el cuerpo en el mismo fichero. Esto también es contrario a las reglas del gnat.

Para estas situaciones, existe una herramienta llamada "gnatchop", que permite crear los ficheros con los nombres apropiados, a partir de otros ficheros. Habrá que ejecutar:

```
gnatchop -w name.a
```

Declaraciones ayacc

UNIVERSIDAD DE CANTABRIA

Declaraciones de los lexemas o “tokens”:

```
%token nombre
```

Declaración de paquetes a utilizar en el paquete tokens

```
%with package_name;
%use package_name;
```

Otras declaraciones a utilizar en el paquete tokens, al menos con el tipo `YYSType`. Este tipo de datos se utiliza para almacenar el dato resultante de cada regla gramatical:

```
{
   type YYSType is ...;
}
```


Notas:

UNIVERSIDAD DE CANTABRIA

Ejemplo de la sección de declaraciones ayacc:

```
%token identificador
%token numero
%token operador_asignacion

%with Var_Strings
%use Var_Strings

{
   type YYSType is record
      Operador : Character;
      Valor : Integer;
   end record
}

%%
```

El paquete XXXX_Tokens se crea con los tokens indicados, y los predefinidos End_Of_Input y Error.

Reglas gramaticales en ayacc

UNIVERSIDAD DE CANTABRIA

Cada regla se describe con un no-terminal, el símbolo ":", una lista de tokens y no-terminales, y las acciones a realizar cuando se detecta la regla:

```
A : B C D ;
A : E F ;
B : H I {acciones};
```

También es posible agrupar varias definiciones para el mismo no-terminal juntas:

```
A  : B C D
   | E F
   ;
B  : H I {acciones};
```

La primera regla es la de comienzo.

Notas:

UNIVERSIDAD DE CANTABRIA

Ejemplo de sección de reglas gramaticales ayacc:

```
linea      : expresion fin_linea
           ;
expresion  : termino
           | expresion '+' termino
           | expresion '-' termino
           ;
termino    : factor
           | termino '*' factor
           | termino '/' factor
           ;
factor     : numero
           | '(' expresion ')'
           ;
```

La expresión y el término se separan en este caso, para hacer que la suma y la resta sean menos precedentes que el producto y la división y todos a su vez menos precedentes que el paréntesis.

Acciones de las reglas gramaticales

UNIVERSIDAD DE CANTABRIA

Cada acción representa instrucciones Ada a realizar al detectarse texto que cumple la regla gramatical asociada.

El valor asociado a cada elemento de la regla (del tipo `YYSType`) se puede representar con un símbolo especial:

```
$$ : valor de la parte izquierda de la regla
$1 : valor del 1º elemento de la parte derecha
$2 : valor del 2º elemento de la parte derecha
...
```

Ejemplo, suponiendo que YYStype es un entero:

```
A : B C D     {$$:=1;};
A : B '+' C   {$$:=$1+$3;};
D : E F       {Put($1);};
```


Notas:

UNIVERSIDAD DE CANTABRIA

Ejemplo de reglas gramaticales con acciones:

```
linea     : expresion fin_linea      {Put($1);New_Line;}
          ;
expresion : termino                  {$$:=$1;}
          | expresion '+' termino    {$$:=$1+$3;}
          | expresion '-' termino    {$$:=$1-$3;}
          ;
termino:  factor                     {$$:=$1;}
          | termino '*' factor       {$$:=$1*$3;}
          | termino '/' factor       {$$:=$1/$3;}
          ;
factor :  numero                     {$$:=YYVal;}
          | '(' expresion ')'        {$$:=$2;}
          ;
```

La variable **YYVal** permite obtener el valor del número, obtenido por el analizador léxico.

Ejemplo de uso de `aflex` y `ayacc`.

UNIVERSIDAD DE CANTABRIA

El siguiente ejemplo es un compilador completo para gestionar un servidor de comunicación de mensajes.

Las acciones a realizar por el servidor llegan en forma de instrucciones de un lenguaje especial, y deben traducirse a operaciones del paquete `Operaciones_Servidor`:

```
destino          =>   "nombre_computador";
envia            =>   "texto_mensaje";
varN             =>   valor_entero;
if varN  envia   =>   "texto_mensaje";
```

La instrucciones, respectivamente, cambian el destino, envían un mensaje, cambian el valor de la variable `N`, o envían un mensaje si la variable `N` es distinta de cero (`N` es un natural).

Notas:

UNIVERSIDAD DE CANTABRIA

Ejemplo de un posible programa a compilar y ejecutar:

```
destino          =>   "pepito";
envia            =>   "hola que tal";
var0             =>   1;
envia            =>   "hola que tal";
if var0  envia   =>   "condicional";
var1             =>   0;
if var1  envia   =>   "condicional no enviada";
envia            =>   "otro";
```

Ejemplo: especificación `lex`

UNIVERSIDAD DE CANTABRIA

```
-- declaraciones

flecha          "=>"
espacio         " "|\t|\n
texto           \"(.)*\"
var             "var"
num             [0-9]+
fin             ";"
inicio_if       "if"
destino         "destino"
envia           "envia"

%%
```


Notas:

UNIVERSIDAD DE CANTABRIA

```
-- reglas de traducción
{flecha}        {return Flecha;}
{texto}         {Text_Buffer:=
                      To_Var_String(YYText(2..YYText'Length-1));
                 return Texto;}
{var}           {return Var;}
{num}           {Num_Buffer:=Integer'Value(YYText);
                 return Num;}
{fin}           {return fin;}
{inicio_if}     {return Inicio_If;}
{destino}       {return Destino;}
{envia}         {return Envia;}
{espacio}       {null;}
.               {return Error;}
%%
```

Ejemplo: especificación `lex` (cont.)

UNIVERSIDAD DE CANTABRIA

```
-- código auxiliar
with Servidor_Mensajes_Tokens;
use Servidor_Mensajes_Tokens;
with Var_Strings; use Var_Strings;
package Servidor is

    function Get_Text_Buffer return Var_String;

    function Get_Num_Buffer return Integer;

    function YYlex return Token;

end Servidor;
```


Notas:

UNIVERSIDAD DE CANTABRIA

```
package body Servidor is
   Text_Buffer : Var_String;
   Num_Buffer  : Integer;
   function Get_Text_Buffer return Var_String is
   begin
      return Text_Buffer;
   end Get_Text_Buffer;

   function Get_Num_Buffer return Integer is
   begin
      return Num_Buffer;
   end Get_Num_Buffer;
##
    -- aquí va YYLex
end Servidor;
```

Ejemplo: especificación ayacc

UNIVERSIDAD DE CANTABRIA

```
-- definiciones
%token          flecha, texto, var, num, fin, inicio_if
%token          destino, envia
%with           Var_strings;
%use            Var_Strings;
{
  type YYstype is record
    varnum,
    num : Integer;
    str : Var_String;
  end record;
}

%%
```


Notas:

UNIVERSIDAD DE CANTABRIA

```
-- reglas gramaticales
programa            : programa instruccion
                    | instruccion
                    ;
instruccion         : inst_envia_cond
                    | inst_var
                    | inst_destino
                    | inst_envia
                    ;
inst_envia_cond     : inicio_if variable envia flecha texto fin
                    {if valor_variable($2.Varnum)/=0 then
                           Envia(Destino => Destino_Actual,
                                 Mensaje => Get_Text_Buffer);
                     end if;}
                    ;
```

Ejemplo: especificación ayacc (cont.)

UNIVERSIDAD DE CANTABRIA

```
inst_var            : set_variable fin
                    {Cambia_Variable($1.Varnum,$1.Num);}
                    ;
set_variable        : variable flecha num
                    {$$.Num:=Get_Num_Buffer;
                     $$.varnum:=$1.varnum;}
                    ;
variable            : var num
                    {$$.varnum:=Get_Num_Buffer;}
                    ;
inst_destino        : destino flecha texto fin
                    {Cambia_Destino(Get_Text_Buffer);}
                    ;
```


Notas:

UNIVERSIDAD DE CANTABRIA

```
inst_envia          : envia flecha texto fin
                    {Envia(Destino => Destino_Actual,
                           Mensaje => Get_Text_Buffer);}
                    ;
%%

with Operaciones_servidor; use Operaciones_Servidor;
with Servidor; use Servidor;
with Var_Strings; use Var_Strings;
with Servidor_Mensajes_Tokens, Servidor_Mensajes_Shift_Reduce,
  Servidor_Mensajes_Goto;
use Servidor_Mensajes_Tokens, Servidor_Mensajes_Shift_Reduce,
  Servidor_Mensajes_Goto;
with Text_IO; use Text_IO;
procedure Servidor_Mensajes is
```

Ejemplo: especificación ayacc (cont.)

UNIVERSIDAD DE CANTABRIA

```
      procedure YYError (S : in String) is
      begin
          Put_Line(S);
      end YYError;

##
      -- aquí va YYParse
begin
     YYparse;
end Servidor_Mensajes;
```


Notas:

UNIVERSIDAD DE CANTABRIA

```
with Var_Strings; use Var_Strings;
package Operaciones_Servidor is

     subtype Num_Variable is Integer range 0..10;

    procedure Envia (Destino : Var_String; Mensaje : Var_String);

     procedure Cambia_Variable (Num : Num_Variable;
                                Valor : Integer);
     function Valor_Variable (Num : Num_Variable) return Integer;

     procedure Cambia_Destino (Destino : Var_String);
     function  Destino_Actual return Var_String;

end Operaciones_Servidor;
```

4. Conclusión

UNIVERSIDAD DE CANTABRIA

Los compiladores para lenguajes de alto nivel tienen muchas fases, en las que progresivamente se va transformando un texto de un lenguaje a otro

Muchas aplicaciones necesitan lenguajes especiales y, por tanto, compiladores

Estos compiladores sencillos se pueden construir con herramientas para análisis léxico y sintáctico:

- **lex o aflex: permiten generar analizadores léxicos**
- **yacc o ayacc: permiten generar analizadores sintácticos**

Con estas herramientas se pueden construir analizadores o "parsers" en C o Ada, y luego integrarlos con la aplicación.

UNIVERSIDAD DE CANTABRIA