Yacc/Bison

Índice

Marina de la Cruz

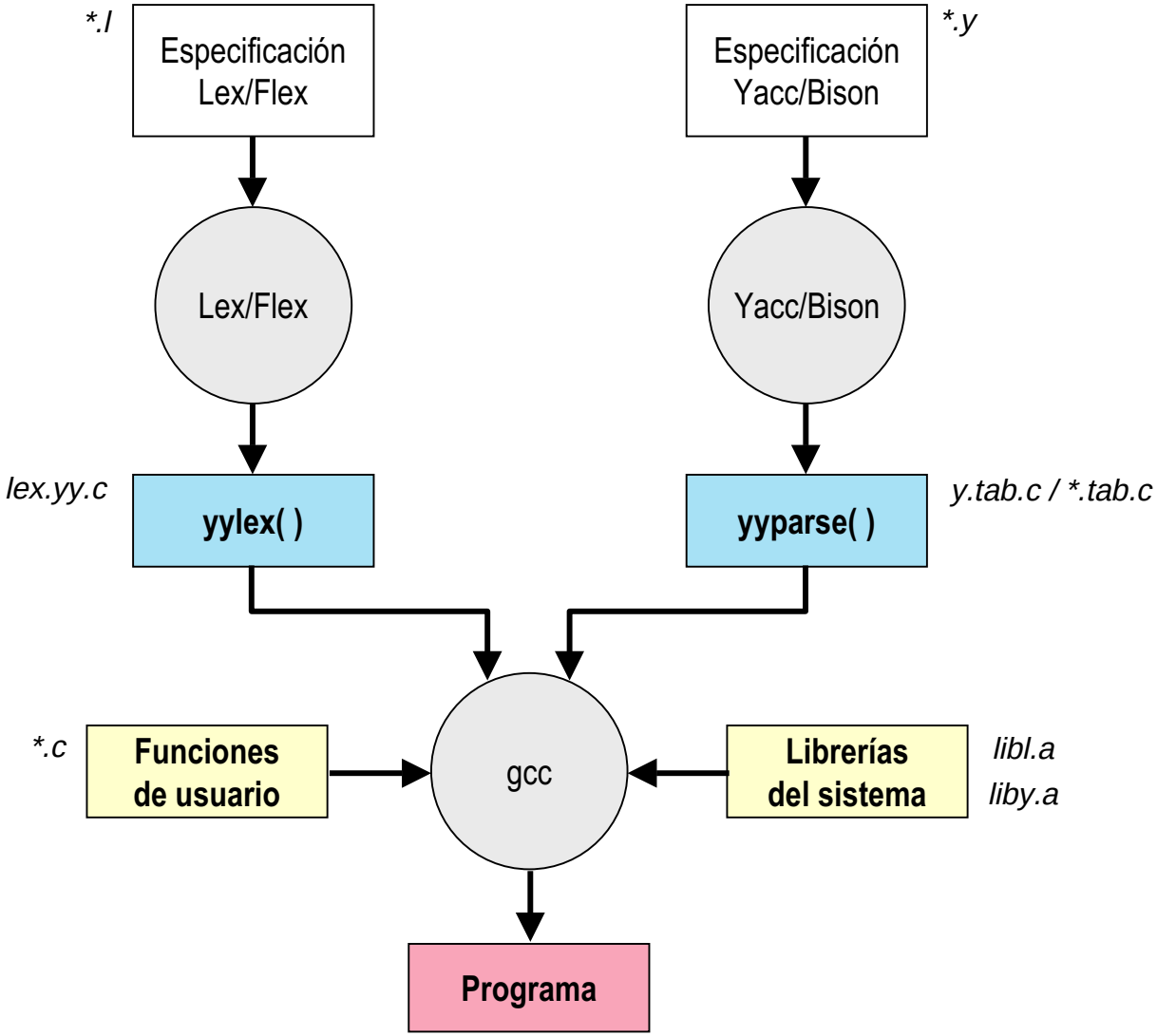
Introducción

- *Yacc* es un generador de analizadores sintácticos LALR(1).
- *Bison* es la versión GNU de *Yacc*.
  - Es totalmente compatible con *Yacc*.
  - Mejora algunas de las prestaciones de *Yacc*.

- Habitualmente *Yacc/Bison* se utiliza conjuntamente con *Lex/Flex*.
  - *Lex/Flex* genera un analizador léxico: **yylex()**
  - *Yacc/Bison* genera un analizador sintáctico: **yyparse()**

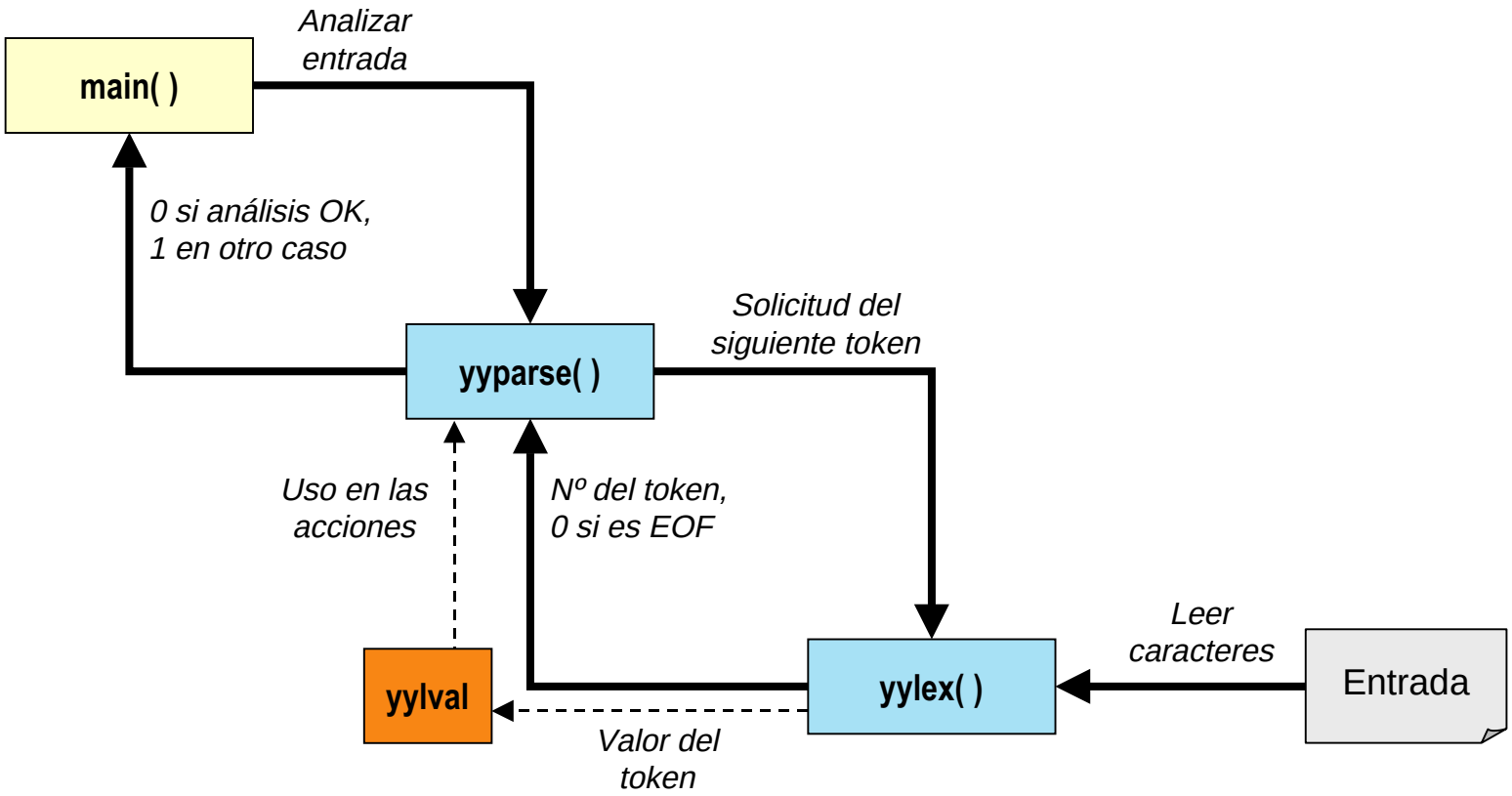
Uso de Yacc/Bison con Lex/Flex (I)

Construcción del programa objetivo



Uso de Yacc/Bison con Lex/Flex (II)

Flujo de control de las funciones yylex() e yyparse()



Formato del fichero de especificación de Yacc/Bison (I)

Estructura del fichero

- Un fichero de especificación de Yacc/Bison tiene cuatro secciones:

```
%{

Declaraciones C

%}

Declaraciones Yacc/Bison

%%

Reglas de la gramática

%%

Código C
```

- Las cadenas “%{“, “%}” y “%%” sirven para delimitar las secciones.
- Se pueden colocar comentarios del tipo /* ... */ en cualquiera de las secciones.

Formato del fichero de especificación de Yacc/Bison (II)

Sección de declaraciones C

- Contiene:
  - definiciones de macros
  - declaraciones de variables
  - declaraciones de funciones
  - directivas `#include`

  ... que se van a utilizar en las acciones de las reglas gramaticales.

- El contenido de esta sección se copia literalmente al principio del fichero que genera Yacc/Bison (y.tab.c/*.tab.c).

Formato del fichero de especificación de Yacc/Bison (III)

Sección de declaraciones Yacc/Bison (I)

- En esta sección se puede declarar el tipo de la variable *yylval*, los símbolos (terminales y no terminales) de la gramática, el axioma de la gramática, y la asociatividad y precedencia de los operadores.

- Declaración %union
  - Por defecto, la variable *yylval* mediante la cual Lex/Flex le pasa a Yacc/Bison los valores semánticos de los tokens es de tipo “int”. Pero habitualmente, los valores semánticos de los tokens son de distintos tipos. Por ejemplo, un token de tipo identificador (ID) tiene un valor semático de tipo “char*”, mientras que un token de tipo constante numérica (NUM) tiene un valor semántico de tipo “int”. Con la declaración %union se define indirectamente una unión de C con un campo para cada tipo de valor semántico, por ejemplo:

```
%union
    {
     char* cadena;
     int numero;
    }
```

Formato del fichero de especificación de Yacc/Bison (IV)

Sección de declaraciones Yacc/Bison (II)

- Declaración %token (i)
  - Se utiliza para definir los simbolos no terminales (tokens) de la gramática.
  - La forma más sencilla es:

```
%token NOMBRE_TOKEN
```

  - Una forma más completa es:

```
%token <campo_union> NOMBRE_TOKEN
```

  - Ejemplo:

```
%token IF
%token THEN
...
%token <cadena> ID
%token <numero> NUM
```

  - Desde Lex/Flex, los tokens cualificados se devolverán haciendo, por ejemplo:

```
[A-Z]+   { yylval.identificador = yytext; return ID; }
[0-9]+   { yylval.numero = atoi(yytext); return NUM; }
```

Formato del fichero de especificación de Yacc/Bison (V)

Sección de declaraciones Yacc/Bison (III)

- Declaración %token (ii)
  - Por convenio, los nombres de los tokens se ponen en mayúsculas.
  - Se pueden agrupar varios token en una linea si tienen en mismo tipo.
  - Los tokens de un único carácter no es necesario declararlos, porque están declarados implícitamente.
    Desde Lex/Flex, este tipo de tokens se devolverán haciendo, por ejemplo:

```
"+"     {return '+'; }
"("     {return '('; }
";"     {return ';'; }
```

  - El número 0 está reservado para el fin de fichero. Lex/Flex hará lo siguiente:

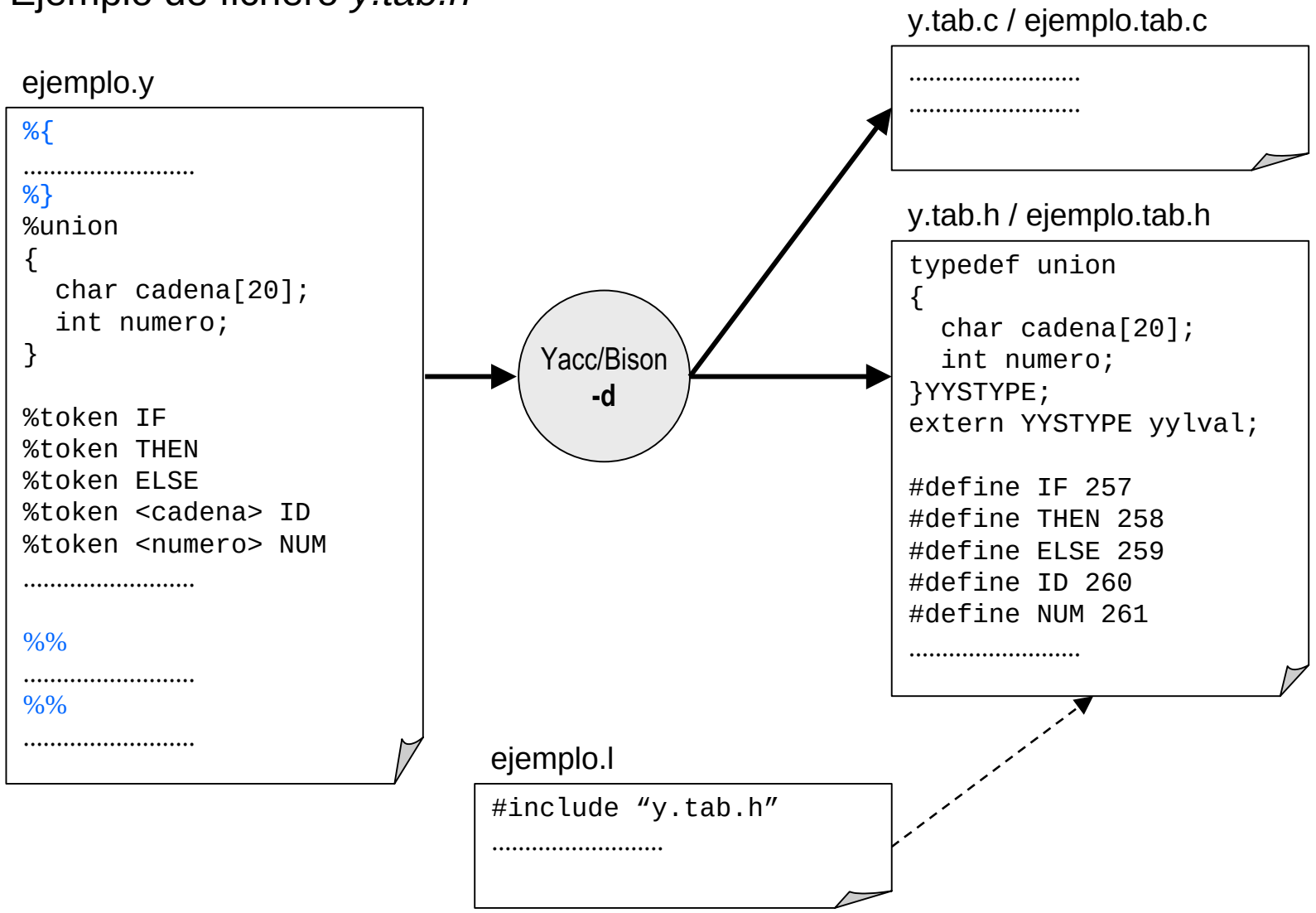
```
<<EOF>>     {return 0; }
```

  - Cuando se compila el fichero de especificación .y se utiliza la opción -d, que fuerza a que Yacc/Bison genere el fichero y.tab.h/*.tab.h que contiene las definiciones de los tokens. Si este fichero se incluye en la especificación de Lex/Flex, se consigue que Yacc/Bison y Lex/Flex compartan las definiciones de tokens.

Formato del fichero de especificación de Yacc/Bison (VI)

Sección de declaraciones Yacc/Bison (IV)

- Ejemplo de fichero *y.tab.h*



Formato del fichero de especificación de Yacc/Bison (VII)

Sección de declaraciones Yacc/Bison (V)

- Declaración %type
  - Se utiliza cuando se ha hecho la declaración %union para especificar múltiples tipos de valores.
  - Permite declarar el tipo de los símbolos no terminales. Lo no terminales a los que no se les asigna ningún valor a través de $$ no es necesario declararlos (ver sección de reglas de la gramática)
  - La declaración es de la forma

```
%type <campo_union> nombre_no_terminal
```

  - Por convenio, los nombres de los no terminales se ponen en minúsculas.
  - Se pueden agrupar varios no terminales en una linea si tienen en mismo tipo.

- Declaración %start
  - Declara cual es el axioma de la gramática.
  - Si se omite la declaración, se asume que el axioma de la gramática es el primer no terminal de la sección de reglas de la gramática.
  - La declaración es de la forma

```
%start axioma
```

Formato del fichero de especificación de Yacc/Bison (VIII)

Sección de declaraciones Yacc/Bison (VI)

- Declaraciones %left y %right
  - Permiten declarar la asociatividad de los operadores de la gramática.
  - La declaración %left especifica asociativiadad por la izquierda.
  - La declaración %right especifica asociativiadad por la derecha.
  - La precedencia de los operadores queda establecida por el orden de aparición de las declaraciones de asociatividad, siendo de menor precedencia en operador que antes se declara.
  - Por ejemplo, las declaraciones:

```
%left '+' '-'
%left '*' '/'
```

    establecen que los cuatro operadores son asociativos por la izquierda, que '+' y '-' son de la misma precedencia, pero de menor que '*' y '/'.

Formato del fichero de especificación de Yacc/Bison (IX)

Sección de reglas de la gramática (I)

- Es la sección más importante del fichero de especificación.
- Contiene las reglas de la gramática escritas en un formato concreto y con acciones asociadas a las reglas opcionalmente.
- Formato de las reglas:
  - `simboloNoTerminal:  simb1 simb2 … simbM  [acción]`
    `;`
  - si varias reglas tienen la misma parte izquierda se pueden agrupar

```
simboloNoTerminal:    parteDerecha1 [acción1]
                    | parteDerecha2 [acción2]
                      ……
                    ;
```

  - por claridad, se comentan las reglas lambda:

```
simboloNoTerminal:    /* vacío */   [acción1]
                    | parteDerecha2 [acción2]
                    ;
```

Formato del fichero de especificación de Yacc/Bison (X)

Sección de reglas de la gramática (II)

- Ejemplo de especificación de reglas (sin acciones):

Descripción BNF del lenguaje

```
……….

<declaration>  ::= <mode> <idlist>

……….

<mode>     ::= bool
            |  int
            |  ref <mode>

……….

<procedures>  ::= <procedure><procedures>
                |

……..
```

Ejemplo.y

```
……….

declaration:   mode idlist
               ;

……….

mode:        BOOL
           | INT
           | REF mode
           ;

……….

procedures:   procedure procedures
            | /* vacío */
            ;

……..
```

Formato del fichero de especificación de Yacc/Bison (XI)

Sección de reglas de la gramática (III)

- Las acciones de las reglas:
  - las acciones son un conjunto de instrucciones C encerradas entre llaves, y que se ejecutan cada vez que se reconoce una instancia de la regla.
  - Normalmente se sitúan al final de la regla, aunque también se admiten en otras posiciones de la regla.
  - La mayoría de las acciones trabajan con los valores semánticos de los símbolos de la parte derecha, accediendo a ellos mediante pseudo-variables del tipo $N, donde N representa la posición del símbolo. El valor semántico del símbolo no terminal de la parte izquierda de la regla se referencia con $$.
    El tipo del valor semántico de un símbolo es el que se le haya asociado mediante una declaración %token (terminales) o %type (no terminales).
  - La acción por defecto es:

```
$$ = $$1
```

  - Ejemplo:

```
exp:     ...
       | exp ‘+’ exp {$$ = $1 + $3}
       ;
```

Formato del fichero de especificación de Yacc/Bison (XII)

Sección de código C

- En esta sección se ubican todas las funciones de soporte.

- Hay tres funciones que siempre tienen que aparecer:
  - **main**: función principal que invoca a la función *yyparse* que realiza el análisis sintactico.
  - **yylex**: función invocada desde *yyparse* para obtener un nuevo token de la entrada.
  - **yyerror**: función invocada desde *yyparse* cuando ocurre un error sintáctico.

- Estas tres funciones se pueden insertar en esta sección o en un fichero aparte para ser linkado posteriormente.

Notificación y recuperación de errores (I)

Generalidades

- Distinguimos entre notificación y recuperación de errores.

- **Notificación**
  - información al exterior de que ha ocurrido un error en el análisis.

- **Recuperación**
  - mecanismo que permita continuar el análisis después de que haya ocurrido un error.
  - **Ventajas**: mejora la productividad del programador porque reduce el tiempo del ciclo programar-compilar-corregir.
  - **Inconvenientes**: puede desencadenar errores en cascada.

- La primera tarea que hay que hacer es decidir cuáles son los errores que el analizador sintáctico tiene que notificar y ante qué errores se podrá recuperar.

Notificación y recuperación de errores (II)

Notificación de errores (I)

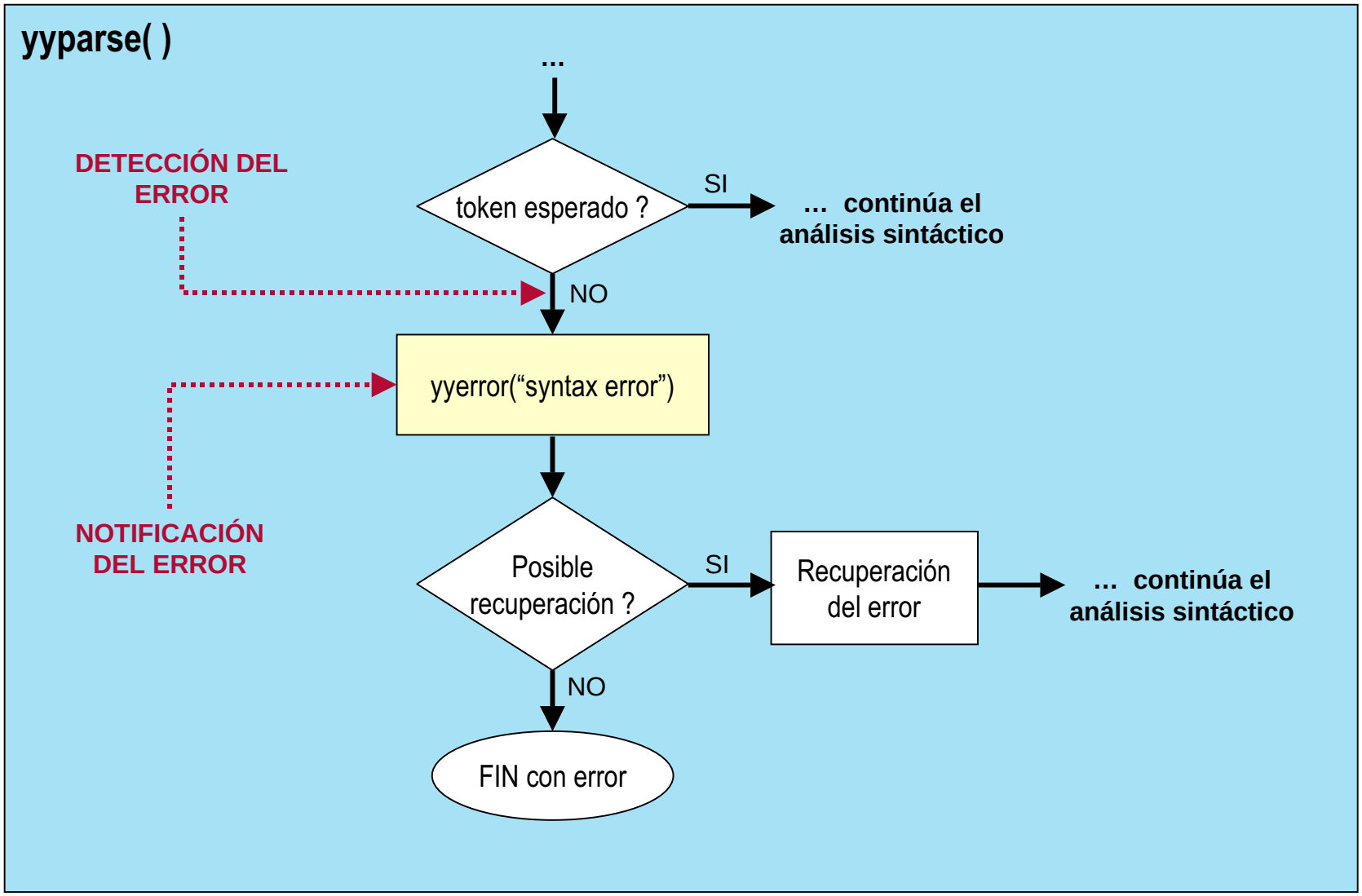
- Se produce un error cuando el analizador sintáctico, *yyparse*, recibe del analizador léxico, *yylex*, un token que no puede satisfacer ninguna producción de la gramática.
- Cuando *yyparse* detecta un error, para notificarlo, invoca la función *yyerror* que debe proporcionar el usuario.
- La forma más sencilla de la función *yyerror* es:

```
void yyerror(char* msg)
{
 fprintf(stderr, "%s\n", msg);
}
```

- El mensaje con el que se invoca la función *yyerror* es "syntax error" (demasiado genérico aunque se acompañe con el número de línea donde se ha producido el error).
- Cuando finaliza la función *yyerro*r, *yyparse* intenta la recuperación del error (siempre que sea posible, ver apartado de recuperación de errores). Y si la recuperación es imposible, *yyparse* termina devolviendo un 1.

Notificación y recuperación de errores (III)

Notificación de errores (II)



Notificación y recuperación de errores (IV)

Notificación de errores (III)

- El analizador léxico puede colaborar en la implementación de una buena política de notificación de errores.
  - **Ejemplo**: si se quiere limitar la longitud de los identificadores, lo natural es controlar esta longitud en el analizador léxico, por ejemplo, de la siguiente manera:

```
[A-Z]+  { yylval.identificador = yytext;
          if strlen(yylval.identificador)>MAXID)
               return ERROR;
          return ID; }
```

Si analizador sintáctico espera recibir un token ID y recibe un token ERROR llamará a la función *yyerror* que mostrará en mensaje “syntax error” sin ninguna indicación del problema que ha ocurrido. Se puede mejorar la notificación del error de la siguiente manera:

```
[A-Z]+   { yylval.identificador = yytext;
           if strlen(yylval.identificador)>MAXID)
                fprintf(stderr, “ID demasiado largo”);
           return ID; }
```

La técnica de aceptar entradas “ilegales” y comprobarlas es una técnica muy potente para mejorar la notificación de errores de un compilador.

Notificación y recuperación de errores (V)

Recuperación de errores (I)

- Recuperación de errores en Yacc/Bison:
  - Se puede recuperar errores utilizando el token especial “error” que está siempre definido y reservado para la gestión de errores.
  - El token “error” se utiliza para construir nuevas reglas gramaticales, situándolo en las posiciones donde se esperan errores.
  - Por ejemplo:

```
sentencias:      /* vacío */
               | sentencias ';'
               | sentencias exp ';'
               | sentencias error ';'
               ;
```

La cuarta regla se puede leer como: un token ‘error’ seguido de ‘;’ es una construcción válida despues de ‘sentencias’.
¿Qué ocurre cuando se produce un error en la mitad de ‘exp’?. El analizador saca elementos de la pila de análisis hasta que llega a un estado en el que el token ‘error’ es legal (en este caso ‘sentencias’), y avanza la entrada hasta encontrar un token legal (en este caso ‘;’).

Notificación y recuperación de errores (VI)

Recuperación de errores (II)

- Las reglas que incorporan el token “error” también pueden tener acciones asociadas.

- La recuperación de errores puede provocar la generación de errores en cadena, y para evitar esta situación, después de un error, el analizador no muestra nuevos mensajes de error hasta que tres tokens consecutivos de la entrada hayan sido desplazados correctamente. Llamando a *yyerrok* se recuperan los mensajes de error inmediatamente. La llamada se puede hacer en las acciones asociadas a la regla.

- El token de entrada que provocó el error se puede eliminar de la entrada llamando a la función *yyclearin*. La llamada se puede hacer en las acciones asociadas a la regla.