

Curso: Compiladores  
Tema 6. Traducción dirigida por Sintaxis  
(Parte 3)

## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis Semántico a través del Análisis Sintáctico.

En el ejercicio de la clase pasada obtuvimos la tabla de Parser para un análisis predictivo. Ahora veamos cómo serían las funciones en un análisis recursivo.

#### **Análisis semántico y traducción LL recursivo**

La gramática en la cual se basa el análisis semántico y traducción es una gramática de traducción con atributos, donde tanto los símbolos de acción como los atributos juegan un papel esencial; por lo que las funciones para realizar dichas tareas en un análisis recursivo involucran estos elementos de forma sustancial.

Recordemos que en el Tema 4. Análisis Sintáctico Descendente, las funciones se construían, una por cada no-terminal y de acuerdo al tipo de producción.

Las funciones no contenían argumentos ni tenían tipo. Ahora dependiendo del tipo de atributos (heredados o sintetizados) que tengan las producciones, las funciones podrán tener argumentos y/o tipo. A través del siguiente ejercicio, conoceremos cómo actúan los atributos en cada tipo de producción.

Caso	Tipo producción	Código
1	$i : A \rightarrow b$	<code>c= getchar( )</code> <code>return</code>
2	$i : A \rightarrow b\alpha$	<code>c= getchar( )</code> <code>procesar(<math>\alpha</math>)</code> <code>return</code>
3	$i : A \rightarrow \xi$	<code>return</code>
4	$i : A \rightarrow B\alpha$	<code>procesar(<math>B\alpha</math>)</code> <code>return</code>

## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis semántico y traducción LL recursivo

#### Ejercicio 6.4.1.

Obtener las funciones del analizador semántico recursivo de la siguiente gramática.

$$1: D \rightarrow T_{\text{t}} a_{\text{p}} \{AT\}_{\text{p1,t1}, V_{\text{t2}}} \left\{ \begin{array}{l} \text{p1} \leftarrow \text{p} \\ \text{t1} \leftarrow \text{t} \\ \text{t2} \leftarrow \text{t} \end{array} \right.$$

$$2: T_{\text{t1}} \rightarrow i_{\text{t}} \quad \left\{ \begin{array}{l} \text{t1} \leftarrow \text{t} \end{array} \right.$$

$$3: T_{\text{t1}} \rightarrow f_{\text{t}} \quad \left\{ \begin{array}{l} \text{t1} \leftarrow \text{t} \end{array} \right.$$

$$4: T_{\text{t1}} \rightarrow c_{\text{t}} \quad \left\{ \begin{array}{l} \text{t1} \leftarrow \text{t} \end{array} \right.$$

$$5: V_{\text{t}} \rightarrow , a_{\text{p}} \{AT\}_{\text{p1,t1}} V_{\text{t2}} \left\{ \begin{array}{l} \text{p1} \leftarrow \text{p} \\ \text{t1} \leftarrow \text{t} \\ \text{t2} \leftarrow \text{t} \end{array} \right.$$

$$6: V_{\text{t}} \rightarrow ;$$

■ Atributo sintetizado  
■ Atributo heredado

## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis semántico y traducción LL recursivo

#### Ejercicio 6.4.1.

Respuesta

Función para D

1:  $D \rightarrow T_t a_p \{AT\}_{p1,t1}, V_{t2}$

{

$p1 \leftarrow p$

$t1 \leftarrow t$

$t2 \leftarrow t$

Atributo sintetizado

Atributo heredado

El símbolo inicial de la gramática D no cuenta con ningún atributo, por lo que la función no tiene ni tipo ni argumentos.

D( )

```

{
    tipo= T( );
    if (car == 'a'){
        pos=getValorToken();
        car= getchar();
        token = getToken();
    } else {
        error();return();
    }
    AsignaTipo(tipo,pos);
    V(tipo);
}

```

{

como el no terminal T tiene un atributo sintetizado, su función retornará el valor de ese atributo. En este caso es el *tipo*.

{

se obtiene el campo *valor* del token que se apunta, esto antes de leer el siguiente token.

{

se deben leer a la par el siguiente carácter de la cadena de átomos y el siguiente token.

{

función que actualiza el campo *tipo* en la posición *pos* de la tabla de símbolos

{


como V tiene un atributo heredado, cuando se llame a su función, ya se tiene el valor de ese atributo, por lo que se envía como argumento.


## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis semántico y traducción LL recursivo.

#### Ejercicio 6.4.1.

Función para T

2:  $T_{t1} \rightarrow i_t$      $\left\{ \begin{array}{l} t1 \leftarrow t \end{array} \right.$      Atributo sintetizado

3:  $T_{t1} \rightarrow f_t$      $\left\{ \begin{array}{l} t1 \leftarrow t \end{array} \right.$      Atributo heredado

4:  $T_{t1} \rightarrow c_t$      $\left\{ \begin{array}{l} t1 \leftarrow t \end{array} \right.$

Observamos que el no-terminal del lado izquierdo tiene un atributo sintetizado cuyo valor lo obtiene de un sintetizado del lado derecho; por lo que la función  $T()$  retornará dicho valor. Además vemos que las tres producciones de  $T$  son muy similares, lo único que cambia es el terminal del lado derecho, por tanto, en una sola condición, podemos preguntar por los tres átomos:



```
int T( ){
    if (car=='i' || car=='f' || car=='c'){
        tipo=getValorToken();
        car= getchar();
        token = getToken();
    }else{
        error(); }
    return (tipo);
}
```

$\left\{ \begin{array}{l} \text{se obtiene el campo } \textit{valor} \text{ del token que se apunta, esto antes de leer el siguiente token.} \\ \text{se deben leer a la par el siguiente carácter de la cadena de átomos y el siguiente token.} \end{array} \right.$

## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis semántico y traducción LL recursivo.

#### Función para V

5:  $V_t \rightarrow , a_p \{AT\}_{p1,t1} V_{t2}$        $\left\{ \begin{array}{l} p1 \leftarrow p \\ t1 \leftarrow t \\ t2 \leftarrow t \end{array} \right.$        Atributo sintetizado  
6:  $V_t \rightarrow ;$        Atributo heredado

Observamos que el no-terminal del lado izquierdo tiene un atributo heredado cuyo valor lo obtuvo cuando fue llamado por otra función con el valor del atributo como argumento.

V(int tipo){

  if (car==';'){

    car= getchar();

    token = getToken();

    { se deben leer a la par el siguiente carácter  
      de la cadena de átomos y el siguiente token.

    if (car == 'a'){

      pos=getValorToken();

      { se obtiene el campo *valor* del token que se apunta, esto antes de  
        leer el siguiente token.

      car= getchar();

      token = getToken();

      { se deben leer a la par el siguiente carácter  
        de la cadena de átomos y el siguiente token.

    } else {

      error();return();}

    AsignaTipo(tipo,pos);

    { función que actualiza el campo *tipo* en la posición *pos* de la tabla de símbolos

    V(tipo);

    { como V tiene un atributo heredado, cuando se llame a su función, ya  
      se tiene el valor de ese atributo, por lo que se envía como argumento.

  } else if (car==';'){

    car= getchar();

    token = getToken();

    { se deben leer a la par el siguiente carácter  
      de la cadena de átomos y el siguiente token.

  else{

    error(); }

}

## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis semántico y traducción LL recursivo.

#### *Actividad 6.4.1*

La siguiente gramática define las sentencias declarativas del lenguaje del analizador sintáctico que entregaron. En la Actividad 6.3.2, introdujeron los símbolos de acción y atributos para que actualice el campo *tipo* en la tabla de símbolos, entonces con base en dicha gramática de traducción con atributos, codificar en cuasi C, las funciones de  $D'$ ,  $D$ ,  $L$ ,  $C$ ,  $V$  y  $G$  de su Analizador recursivo.

$$D' \rightarrow DD'$$
$$D' \rightarrow \xi$$
$$D \rightarrow V L:$$
$$L \rightarrow aGC$$
$$C \rightarrow ,L$$
$$C \rightarrow \xi$$
$$V \rightarrow b$$
$$V \rightarrow c$$
$$V \rightarrow f$$
$$V \rightarrow n$$
$$V \rightarrow g$$
$$G \rightarrow [e]$$
$$G \rightarrow \xi$$

## 6. Traducción Dirigida por Sintaxis

### 6.4 Análisis semántico y traducción LL recursivo.

#### *Actividad 6.4.2*

De la gramática definida para el análisis sintáctico que entregaron, analiza qué producciones deben incluir atributos para que realice el análisis semántico de las siguientes tareas:

- Cada vez que encuentre una variable, revisar que esté declarada.
- Que una misma variable no esté declarada dos o más veces.



## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### **Generadores de Parser Yacc y Bison.**

Un generador de Parser muy popular es el llamado *Yacc*, de las siglas “yet another compiler-compiler”. Es un generador de parser LALR, ampliamente disponible; en sistemas UNIX se invoca como un comando y ha sido utilizado como herramienta para la implementación de cientos de compiladores. En diversas distribuciones de Linux, está definido este compilador como *Bison*, que es prácticamente una versión de Yacc.

Yacc y Bison se emplean para generadores de analizadores sintácticos LALR(1), pero además permiten realizar actividades de análisis semántico y traducción dirigida por la sintaxis. Esto nos facilita mucho la construcción de analizadores y traductores basados en gramáticas LR que son mucho más complejas que las LL(1).

El proceso para elaborar el analizador y/o traductor es, primeramente, crear un archivo con la extensión “.y” que contenga las especificaciones de la gramática de traducción. Este archivo se compila con el comando yacc o bison:

`$yacc parser.y`      o      `$bison parser.y`

Se genera un archivo escrito en lenguaje C llamado <nombreArchivo>.tab.c el cual tiene implementado el Parser LALR(1) y otras rutinas de C que el usuario diseñó para completar la función del Parser que se está construyendo. Finalmente este archivo se compila con un compilador de C, generalmente gcc.

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### **Generadores de Parser Yacc y Bison.**

Un programa escrito con Yacc o Bison, tiene la siguiente estructura:

```
<definiciones>
%%
<reglas>
%%
<funciones auxiliares>
```

La sección de *definiciones* contiene información acerca de los tokens y tipos de datos que Yacc necesita para construir un analizador sintáctico y traducción. También incluye cualquier código en C que debería ir directamente en el archivo de salida, por ejemplo las directivas *#include*.

La sección de *reglas* contiene reglas gramaticales o producciones en una forma BNF modificada, junto con acciones en código C que se ejecutarán siempre que se reconozca la regla gramatical asociada (es decir se emplee en una *reducción* de acuerdo con el algoritmo de análisis sintáctico LALR(1)).

Las convenciones de metasímbolos utilizadas en las producciones son de la siguiente manera:

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

Como es habitual, la barra vertical (|) se utiliza para separar las producciones de un mismo no-terminal; el símbolo de flecha ( $\rightarrow$ ) que se usa para separar el lado izquierdo del derecho de una producción se reemplaza por el signo dos puntos (:); finalmente el punto y coma (;) debe estar al final de la última producción del no-terminal.

La tercera sección, de *funciones auxiliares*, contiene declaraciones de procedimientos y funciones que crea el programador porque son necesarias para completar el analizador sintáctico y/o traductor.

Veamos, a través de un ejemplo clásico, cómo se construye un programa en Yacc.

#### *Ejemplo de uso de Yacc/Bison*

El ejemplo es una calculadora de expresiones aritméticas enteras simples. Su gramática es:

1:  $exp \rightarrow exp + term$

2:  $exp \rightarrow exp - term$

3:  $exp \rightarrow term$

4:  $term \rightarrow term * factor$

5:  $term \rightarrow factor$

6:  $factor \rightarrow n$

7:  $factor \rightarrow (exp)$

$N = \{ exp \ term \ factor \}$

$\Sigma_G = \{ n \ + \ - \ * \ ( \ ) \}$

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

Como es una gramática para un análisis LALR debe cumplir con las condiciones de una gramática LR, por lo que debemos incluir un nuevo no-terminal como símbolo inicial de la gramática para que tenga una sola producción:

0:  $result \rightarrow exp$

4:  $term \rightarrow term * factor$

1:  $exp \rightarrow exp + term$

5:  $term \rightarrow factor$

$N = \{ result \ exp \ term \ factor \}$

2:  $exp \rightarrow exp - term$

6:  $factor \rightarrow n$

$\Sigma_G = \{ n \ + \ - \ * \ ( \ ) \}$

3:  $exp \rightarrow term$

7:  $factor \rightarrow (exp)$

El analizador sintáctico y traductor, además de revisar la expresión aritmética de enteros de la entrada, calculará su valor. Por ejemplo, si la entrada es  $(12-10)*22$ , deberá escribir como resultado 44.

Entonces incluiremos los símbolos de acción y sus atributos en la gramática.

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

0:  $result \rightarrow exp_v \{EscrResult\}_r$       $r \leftarrow v$

1:  $exp_{v1} \rightarrow exp_v + term_t$       $v1 \leftarrow v + t$

2:  $exp_{v1} \rightarrow exp_v - term_t$       $v1 \leftarrow v - t$

3:  $exp_v \rightarrow term_t$       $v \leftarrow t$


4:  $term_{t1} \rightarrow term_t * factor_f$       $t1 \leftarrow t * f$

5:  $term_t \rightarrow factor_f$       $t \leftarrow f$

6:  $factor_f \rightarrow n_n$       $f \leftarrow n$

7:  $factor_f \rightarrow ( exp_v )$       $f \leftarrow v$

 Atributo sintetizado

 Atributo heredado

En Yacc/Bison cada elemento terminal, no-terminal y símbolo de acción puede tener no más de un solo atributo sintetizado. Los símbolos de acción pueden tener uno o más atributos heredados.

Los valores de los atributos asociados a los terminales y no-terminales se accederán por medio de su posición dentro de la producción, antecediendo el signo de pesos o dólar (\$), esto para los que se encuentran del lado derecho. El atributo del lado izquierdo se referenciará con \$\$.

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

Entonces veamos cómo se codifica la gramática de ejemplo en la sección de <reglas>:

```
result : exp {printf("%d\n", $1);}
```

```
; /* aquí $1 es el valor del atributo asociado a exp. Implícitamente se hace  $r \leftarrow v$  */
```

```
exp : exp '+' term    {$$=$1+$3;}
```

```
    | exp '-' term    {$$=$1-$3;}
```

```
    | term    {$$=$1;}
```

```
;
```

```
term : term '*' factor {$$=$1*$3;}
```

```
    | factor    {$$=$1;}
```

```
;
```

```
factor : NUM    {$$=$1;}
```

```
    | '(' exp ')' {$$=$2;}
```

```
;
```

1:  $exp_{v1} \rightarrow exp_v + term_t$      $v1 \leftarrow v + t$

$\$ \$$      $\$ 1$      $\$ 3$      $\{ \$ \$ = \$ 1 + \$ 3 \}$

Las reglas de asignación se aplican con acciones (operaciones entre llaves). Obsérvense las posiciones de los no-terminales y terminales que cuentan con valor/atributo asociado.

Si un terminal o no-terminal no tiene asociado ningún valor/atributo, de todas formas cuenta su posición en el lado derecho de las producciones.

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

Revisemos a más detalle algunas producciones:

De las producciones de *factor*:

6:  $factor_f \rightarrow n_n$                        $f \leftarrow n$

7:  $factor_f \rightarrow ( exp_v )$                        $f \leftarrow v$

Su codificación es:

```
factor : NUM      {$$=$1;}  
       | '(' exp ')' {$$=$2;}  
       ;
```

El no terminal del lado izquierdo se coloca en la columna 1. Como son dos producciones, van separadas por '|' y la segunda producción, por ser la última, se finaliza con ';'. Se recomienda alinear : | y ;

Los nombres de los no-terminales siguen las mismas reglas de un identificador de C, pero las buenas prácticas recomiendan sólo usar letras minúsculas.

De los terminales, si están formados por un solo carácter y no tienen asociado ningún valor/atributo, sólo se encierran con apóstrofes; es el caso de la producción 7 con los paréntesis. De los terminales que sí tienen asociados un valor/atributo, deben ser declarados como %token en la sección de <definiciones>, se acostumbra nombrarlos con puras mayúsculas para distinguirlos de los no-terminales. Para este ejemplo:

```
%token NUM
```

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

Vayamos ahora a la sección de funciones auxiliares.

La función *main* debe llamar a la función *yyparse* que es la que se genera al compilar el programa .y :

```
main( )  
{  
    yyparse( );  
}
```

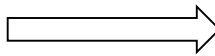
Cuando se ejecuta *yyparse( )*, éste llama a la función *yylex( )* para que le vaya entregando token por token que va generando el analizador léxico.

La función *yylex( )* puede ser externa (en un archivo aparte) o bien definirse en la sección de funciones. Si está de forma externa, ésta es la forma de generar el ejecutable:

Pasos para obtener el programa ejecutable en Linux (respetando el orden):

a) Compilar el programa escrito en yacc/bison

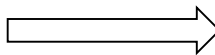
```
$bison -d parser.y
```



Con esto se generan los archivos  
parser.tab.c y parser.tab.h

b) Compilar el programa escrito en lex/flex

```
$flex anlex.l
```



Con esto se genera el archivo  
lex.yy.c

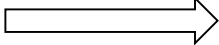


## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

c) Compilar los programas generados en C

`$gcc parser.tab.c lex.yy.c -lfl`  Con esto se obtiene *a.out*

En cada paso hay algunas observaciones que hay que hacer.

- Al compilar el programa escrito en yacc/bison (`$bison -d parser.y`) se indica con el argumento `-d` que genere un archivo de cabecera (`parser.tab.h`) cuyo contenido es la definición de variables usadas tanto en el analizador léxico como sintáctico y traducción. Tiene el siguiente aspecto:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
extern YYSTYPE yylval;
```

- Antes de compilar el programa escrito en lex/flex hay que “incluir” en dicho programa el archivo de cabecera que se generó en el paso anterior (`parser.tab.h`)

```
#include "parser.tab.h"
```

También es importante que sólo el programa `.y` tenga, en sus funciones auxiliares la función `main( )`. Es decir, el programa `.l` no debe tener la función `main( )`

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### Generadores de Parser Yacc y Bison.

Volvamos a nuestro ejemplo de la calculadora.

Además de definir la función *main* en la sección de funciones auxiliares, definiremos la función *yylex()*, es decir será interna.

```
int yylex(void)
{
    int c;
    while ((c=getchar())==' '); /* elimina blancos */
    if (isdigit(c)) { /* encontró el inicio de un número */
        ungetc(c, stdin); /* regresa al buffer de entrada el dígito leído */
        scanf ("%d", &yylval); /* lee el número completo, lo pone en
                                yyval, que es una variable definida por lex-yacc */
        return (NUM); /* NUM está definido como token y su valor asociado
                        es el de yyval */
    }
    if (c == '\n') return 0; /* hace que se detenga el análisis sintáctico */
    return (c); /* envía un carácter diferente a dígitos o al salto de línea */
}
```

## 6. Traducción Dirigida por Sintaxis

### 6.5. Generadores de Parser.

#### **Generadores de Parser Yacc y Bison.**

Finalmente, en la sección de funciones auxiliares agregamos la función de error.

```
void yyerror(char * s)
{
    fprintf(stderr,"%s\n",s); /* permite la impresión de un mensaje de error */
}
```

Al obtener el programa ejecutable y correrlo, la entrada será a través del teclado, de la siguiente forma en la línea de comandos:

```
$ (10-5)*100-230+10
```

y al darle <enter> deberá dar el resultado.

El programa completo lo pueden encontrar en el apartado Documentos -> Material de apoyo de la plataforma Chamilo.

## 6. Traducción Dirigida por Sintaxis

### 6.5 Generadores de Parser.

#### *Actividad 6.5.1*

Elaborar el programa en yacc/bison del parser de la gramática de traducción con atributos definida en la actividad 6.3.2, para que escriba qué tipo de variables se está declarando. Por ejemplo, si la entrada (desde el teclado) es “na,a:” escriba “Las variables son del tipo entero”. La función yylex () debe estar en la sección de funciones auxiliares.