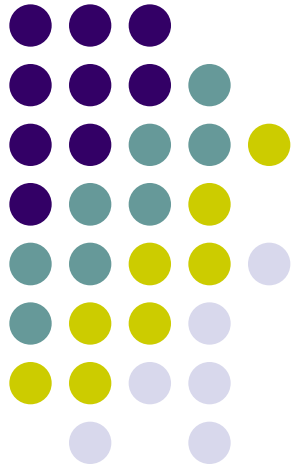
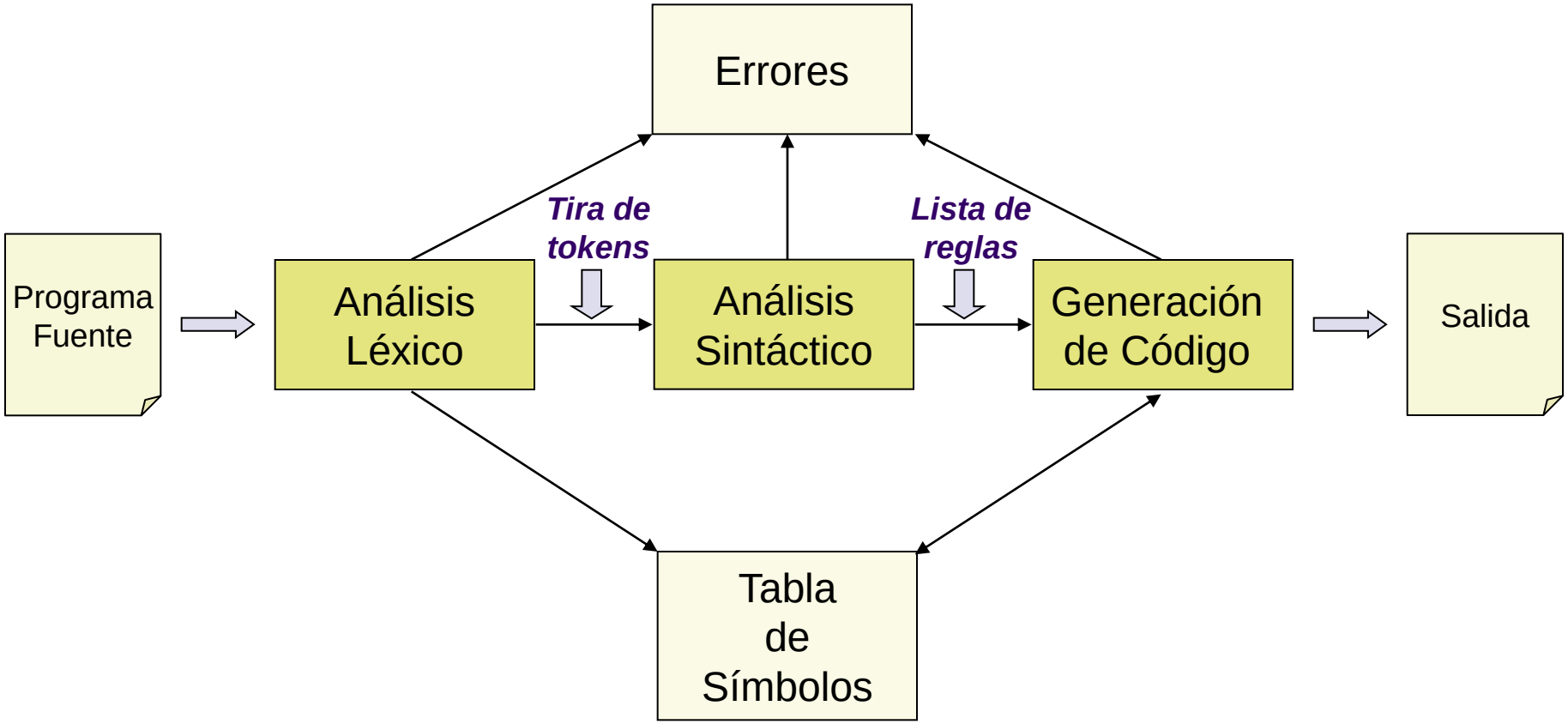
Diseño de Compiladores I

Análisis Sintáctico

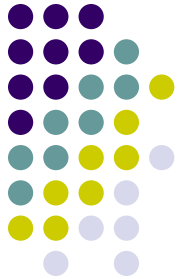
Yacc



Fases de la Compilación

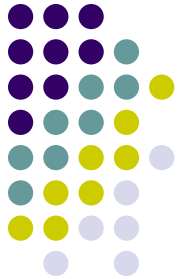


Análisis Sintáctico



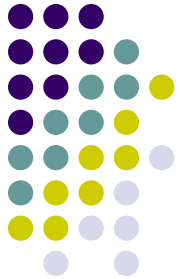
- Agrupa los tokens del programa fuente en frases gramaticales que el compilador usará en las siguientes etapas.
- Obtiene una cadena de tokens del Analizador Léxico, y verifica que la cadena de tokens pueda generarse mediante la gramática del lenguaje.

Estrategias de Parsing



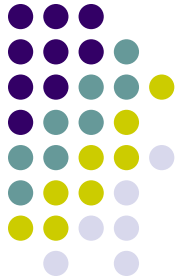
- Parsing Ascendente (botton up)
  - Construye el árbol desde las hojas a la raíz
- Parsing Descendente (top down)
  - Construye el árbol desde la raíz a las hojas

Estrategias de Parsing



- Los Analizadores Sintácticos para la clase más extendida de gramáticas LR (ascendente), se construyen generalmente mediante herramientas.

Parsing Ascendente
Gramáticas LR



- Va del programa a la hipótesis
- El programa se lee de izquierda a derecha (**L**).
- Las reglas se leen de derecha a izquierda (**R**): el lado derecho se reemplaza por el izquierdo.
- Estrategia: ***Reducción***

Parsing Ascendente

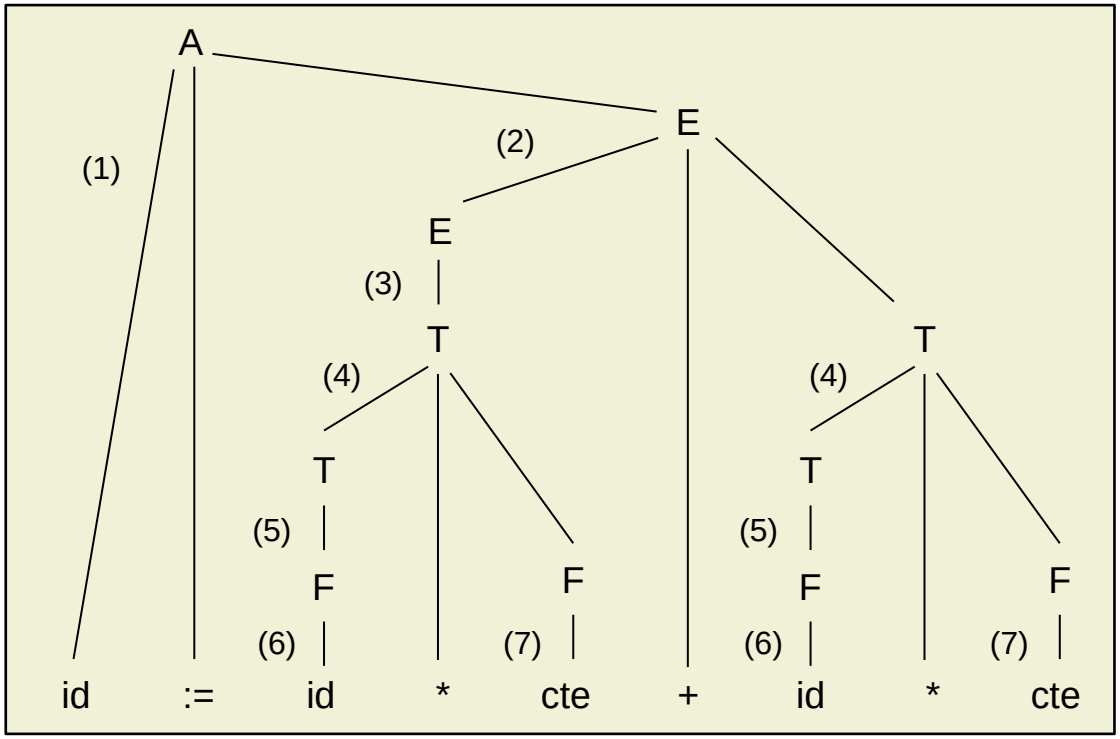
Ejemplo

precio := costo1 * 1.5 + costo2 * 1.2 → id := id * cte + id * cte

Gramática

1) A → id := E
2) E → E + T
3) E → T
4) T → T * F
5) T → F
6) F → id
7) F → cte

Árbol de Parsing

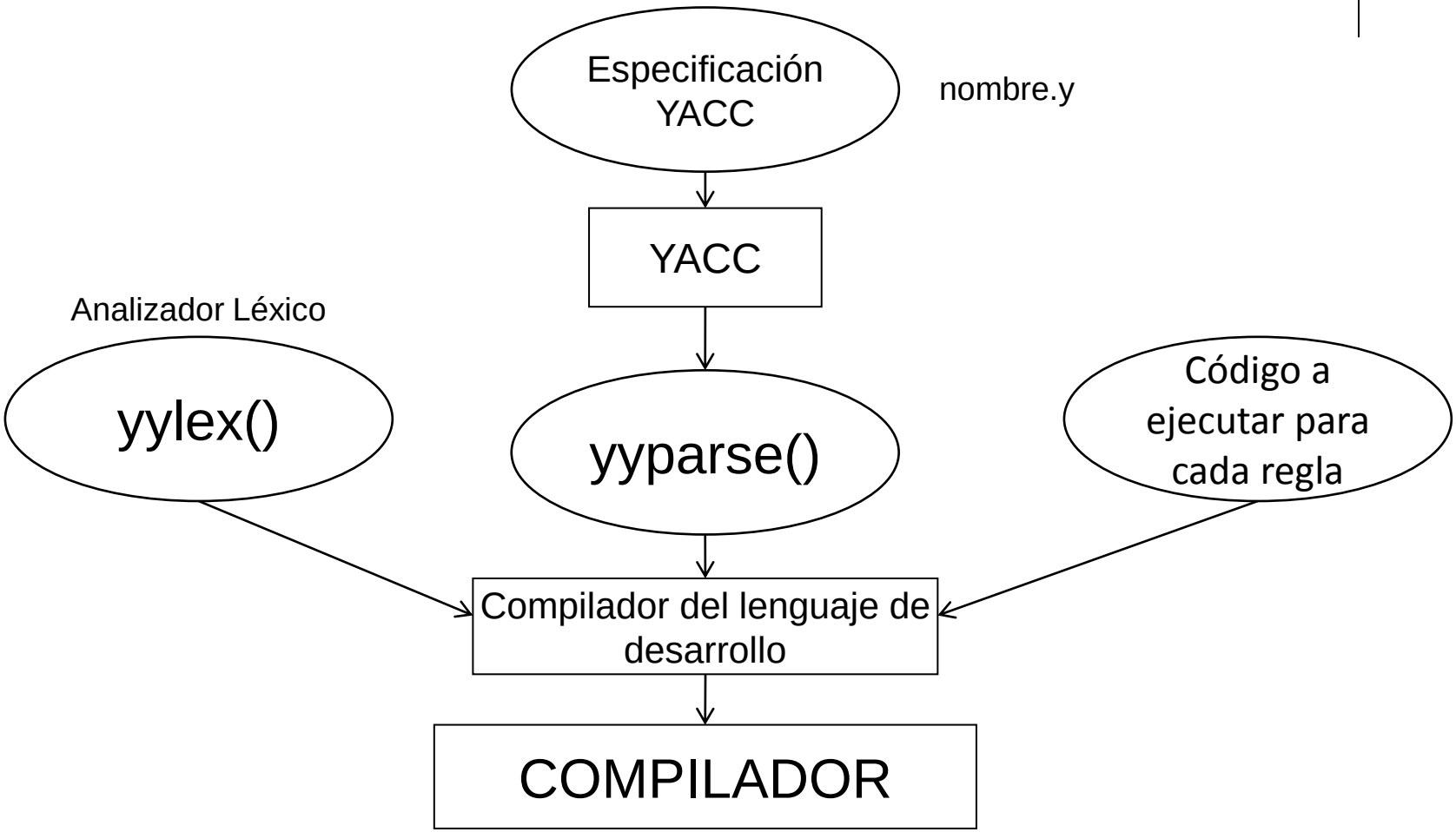


Lista de Reglas: 6 5 7 4 3 6 5 7 4 2 1

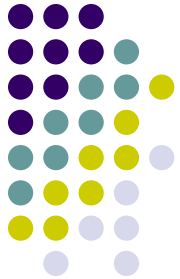
YACC

- YACC provee una herramienta general para analizar estructuralmente una entrada.
- Requiere una especificación que incluye:
  - Un conjunto de reglas que describen los elementos de la entrada (Gramática)
  - Un código a ser invocado cuando una regla es reconocida
  - Un Analizador Léxico que se encargue de proveer tokens

Yacc


Especificación YACC
nombre.y
YACC
Analizador Léxico
yylex()
yyparse()
Código a ejecutar para cada regla
Compilador del lenguaje de desarrollo
COMPILADOR

Usando YACC



- Escribir una especificación YACC que describe la gramática. (.y por convención)
  - Ejemplo: gramatica.y
- Escribir un Analizador Léxico. El método o función léxica debe ser:
  - ***int yylex()***
- Ejecutar YACC con la especificación como parámetro, para generar el código fuente del parser.

  Byacc para C: **byacc gramatica.y**

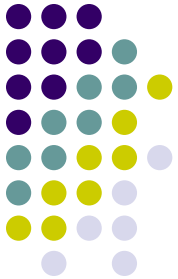
  La salida es un archivo llamado ***y_tab.c*** conteniendo la función ***yyparse***().

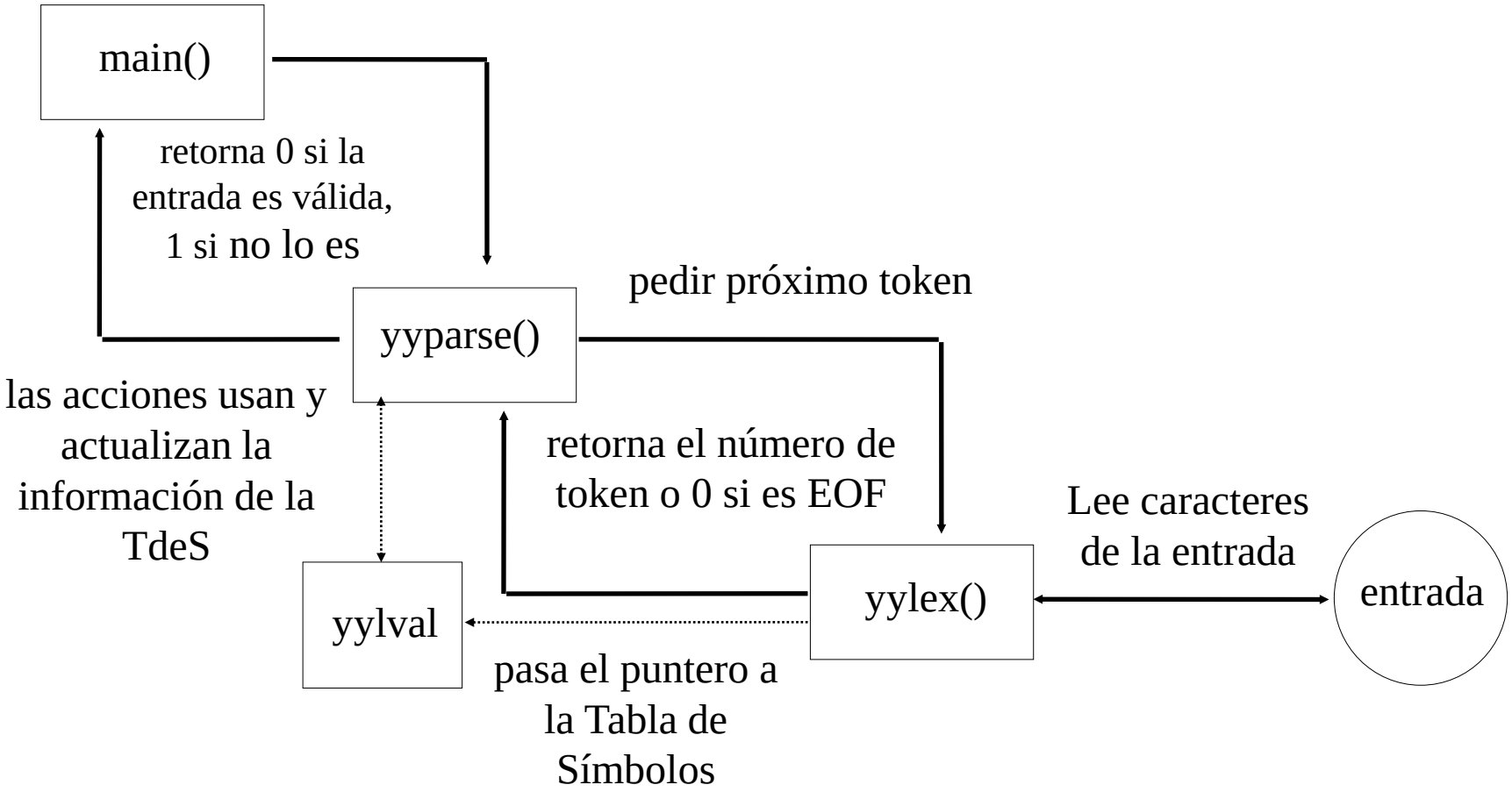
  Byacc para Java: **yacc –J gramatica.y**

  La salida es un archivo llamado ***Parser.java*** conteniendo el método ***yyparse***()

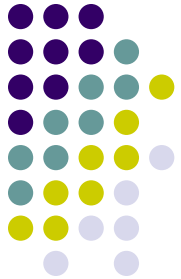
  Nota: ***yyparse*** invoca a ***yylex*** cada vez que necesita un token.
- Compilar y vincular los fuentes del parser, Analizador Léxico y todo otro código creado.

Interacción entre yyparse y el Analizador Léxico



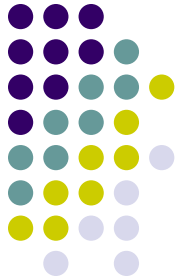


Tokens



- El Analizador Léxico detecta un token y retorna un número de token al parser.
- El número de token es definido por un símbolo que el parser usa para identificar al token.
- Los números de token son definidos por YACC cuando procesa los tokens declarados en la especificación.
- Cada carácter ASCII es definido como un token cuyo número es su valor ASCII (0 a 256).
- Los tokens definidos por el usuario comienzan en 257.

Tokens



- Si, en la especificación para YACC, se incluye:

  %token ID CTE …

- En C, se generan sentencias #define para definir los números de tokens:

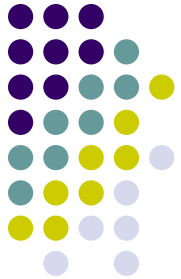
  #define ID 257

  #define CTE 258

  …

- Estas definiciones son ubicadas en el archivo y_tab.c, junto con la rutina **yyparse**, o pueden generarse en un archivo separado, llamado y_tab.h.
  - Para ello, se debe ejecutar byacc –d gramatica.y

Tokens



- Si, en la especificación para YACC, se incluye:

  %token ID CTE …

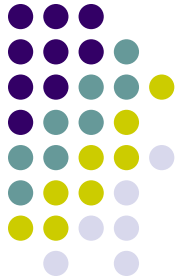
- En Java, en el archivo Parser.Java, junto con yyparse, se generan:

  public final static short ID=257;

  public final static short CTE=258;

  …

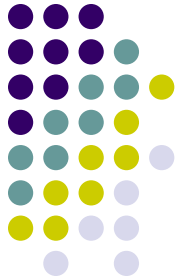
Tokens – yylval
(Yacc para C)



- Para pasar información del token al parser, se utiliza una variable externa llamada **yylval.**
- En C, el tipo de **yylval** es int, por defecto.
- Se puede cambiar el tipo de **yylval** o,
- definir una unión de tipos de datos múltiples para **yylval:**

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

Tokens – yylval
(Yacc para C)



- Desde el Analizador Léxico, se podrá asignar la variable yylval usando:
  - Si se utiliza el tipo int, asignado por defecto a yylval:
    - **yylval = integerval;**
  - Si se define una unión de tipos de datos múltiples:
    - **yylval.entero = integerval;**

    o
    - **yylval.cadena = lexema;**

- Desde el Analizador Sintáctico, se podrá usar el valor asignado:
  - Si se utiliza el tipo int, asignado por defecto a yylval:
    - **$$ = $1;**
  - Si se define una unión de tipos de datos múltiples:
    - **$$.entero = $1.entero + $2.entero;**

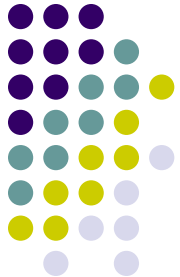
    o
    - **$$.cadena = $1.cadena;**

Tokens – yylval
(Yacc para Java)

- En Java, Yacc genera la clase pública ParserVal

```
public class ParserVal
{
  public int ival;
  public double dval;
  public String sval;
  public Object obj;
  public ParserVal(int val)
  {
    ival=val;
  }
  public ParserVal(double val)
  {
    dval=val;
  }
  public ParserVal(String val)
  {
    sval=val;
  }
  public ParserVal(Object val)
  {
    obj=val;
  }
}//end class
```

Tokens – yylval
(Yacc para Java)



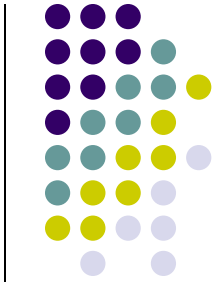
- Desde el Analizador Léxico, se podrá asignar la variable yylval usando:

```
yylval = new ParserVal(doubleval);
yylval = new ParserVal(integerval);
...
o:
yylval = new ParserVal(new myTypeOfObject());
```

- Desde el Analizador Sintáctico, se podrá usar el valor asignado:

```
$$.ival = $1.ival + $2.ival;
$$.dval = $1.dval - $2.dval;
```

Especificación YACC



Gramática:

- Tokens, que son un conjunto de símbolos terminales
- Elementos sintácticos, que son un conjunto de símbolos no terminales
- Reglas de producción
- Una regla **start** que reduce todos los elementos de la gramática a una sola regla.

Especificación YACC

Formato:

declaraciones

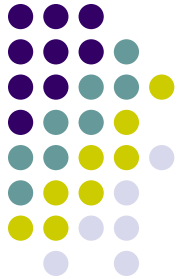
%%

reglas gramaticales

%%

código

Especificación YACC
Sección de Reglas

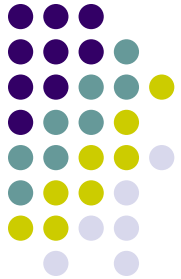


- Formato:

  no terminal : definición {acción}
  ;

  - Un (:) separa el lado izq. del derecho de la regla
  - Un (;) termina la regla. Por legibilidad, el (;) se ubica solo en una línea.
  - La definición consiste de cero o más nombres de terminales, tales como tokens o caracteres literales, y otros símbolos no terminales.
  - Cada definición puede tener una acción asociada, ubicada entre llaves.
  - La barra vertical (|) permite definiciones alternativas dentro de una regla.
  - Los nombres no terminales van en minúsculas y los tokens en mayúsculas por convención.

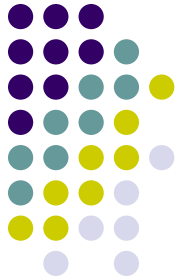
Especificación YACC
Sección de Reglas



- Un carácter literal se encierra entre apóstrofes.
- La \ tiene un significado especial, para secuencias escape:

´\n´	newline
´\r´	return
´\´´	comilla simple
´\\´	barra invertida
´\t´	tab
´\b´	backspace
´\f´	form feed
´*xxx*´	carácter cuyo valor es *xxx*

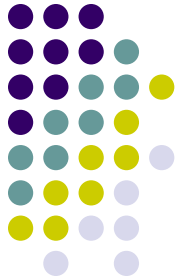
Especificación YACC
Sección de Reglas



Token error

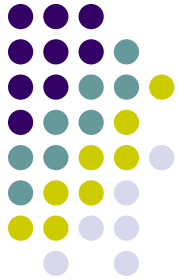
- Se puede usar en las reglas un símbolo llamado ***error***.
- No existe una regla que lo defina, ni se incluye en la declaración de tokens.
- Es un token definido especialmente por Yacc que significa que cualquier token que no aparee ninguna de las otras reglas, apareará la que contiene error.
- Conviene utilizarlo con otro token, que sirve de carácter de *sincronización*. A partir del token erróneo, Yacc tirará todos hasta encontrar ese carácter (por ej. un newline).
- Permite, de algún modo, *recuperar errores*.
- Se puede asociar una acción que permita informar que el token es erróneo, y toda la información que se desee agregar.

Especificación YACC
Sección de Reglas



- Cualquier ***regla*** de la gramática puede tener una ***acción*** asociada
  - Una acción es una o más sentencias en el lenguaje de desarrollo.
  - Frecuentemente, la acción actúa sobre la información de los tokens contenida en la Tabla de Símbolos.

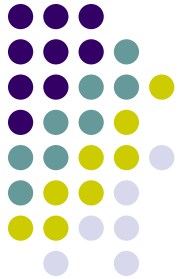
Especificación YACC
Sección de Reglas



- El Analizador Léxico asigna a la variable externa **yylval** el puntero a la entrada en la Tabla de Símbolos donde se almacenó el valor léxico del token.
- YACC provee una notación posicional, $n, para acceder a la información del enésimo token en una expresión:
- Ejemplo:

```
expr:    CTE ´+´ CTE        { valorsuma = getvalor($1) +
                              getvalor($3)); }
```

Acciones de las reglas



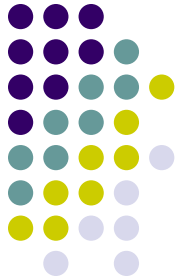
- El valor retornado por una acción puede ser asignado a la variable ¨$$¨:

```
expr        :  ID                          { $$ = $1; }
            |  ´(´ expr ´)´                 { $$ = $2; }
            ;
```

  - La primera acción es invocada cuando se reconoce ¨ID¨, y retorna la información del token **ID** como el valor de la regla expr.

    Esta el la acción por defecto, y puede ser omitida opcionalmente.
  - La segunda acción retorna el valor de expr que es el segundo elemento en la construcción.

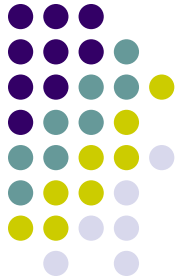
Especificación YACC
Sección de Reglas



Acciones

- El parser generado por YACC guarda los valores de cada token en una variable de trabajo (yyval del mismo tipo que yylval).
- Las variables de trabajo están disponibles para ser usadas dentro de las acciones de las reglas, y son rotuladas $1, $2, $3, etc.
- La pseudo-variable $$ es el valor a ser retornado por esa invocación de la regla.
- En el código real, son reemplazadas con las referencias Yacc correctas.
- Se pueden ejecutar acciones después de cualquier elemento de un conjunto de tokens (no sólo al final)

Especificación YACC
Sección de declaraciones



- **% union** Declara múltiples tipos de datos para los atributos de los tokens (yylval)

  Ejemplo:

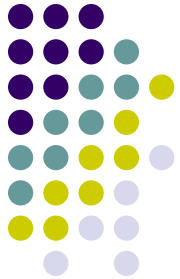
```
% union {
        int entero;
        char *cadena;
      }
```

- **% token** Declara los nombres de los tokens.

  Si se usa union la sintaxis es:

  % token <elem. de la union que corresponde a este grupo de tokens > lista de tokens

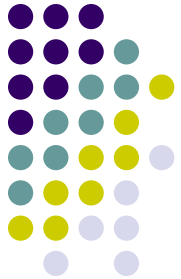
Especificación YACC
Sección de declaraciones



- **% left** Define operadores asociativos a izquierda
- **% right** Define operadores asociativos a derecha

  El orden de estas declaraciones indica la precedencia (de menor a mayor) Si se utiliza esta declaración, no es necesario declarar los operadores como tokens.
- **% nonassoc** Define operadores no asociativos
- **% type** Declara el tipo de los no terminales, cuando se uso union, y en las acciones asociadas a las reglas, se hacen asignaciones a $$
- **% start** Declara el símbolo de start. Por defecto es la primera regla
- **% prec** Asigna precedencia a una regla

Especificación YACC
Sección de declaraciones



- Puede contener código en el lenguaje de desarrollo para declarar variables, tipos, etc.

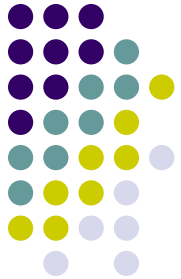
  %{

  declaraciones en lenguaje de desarrollo

  %}

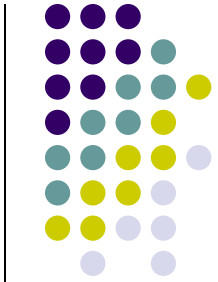
- Cualquier cosa entre %{ y %} es copiada directamente al archivo generado por YACC.

Especificación YACC
Sección de código



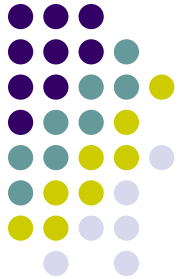
- La sección de código es opcional, pero puede contener cualquier código en el lenguaje de desarrollo provisto por el usuario (incluso el código del Analizador Léxico).

Especificación YACC



- La mínima especificación YACC consiste en una sección de reglas precedidas por una declaración de los tokens usados en la gramática.

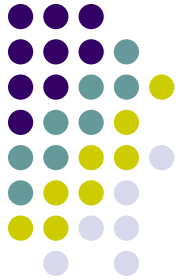
Autómata de Pila



- Los autómatas finitos son suficientes para el Analizador Léxico.
- Los A.F. no son suficientes para un Analizador Sintáctico, porque son incapaces de recordar el *estado anterior*.
- YACC genera un autómata de pila.
- Para obtener el autómata, se debe ejecutar yacc con la opción –v

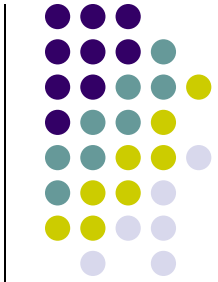
  byacc –v gramatica.y

Autómata de Pila



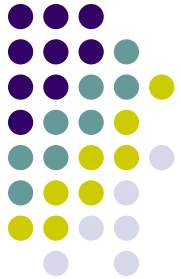
- Tiene un número finito de estados, una función de transición, una entrada, y está equipado con una ***pila***.
- La función de transición trabaja sobre el estado actual, el elemento en el tope de la pila, y el token de entrada actual, produciendo un nuevo estado.
- Permite a YACC reconocer gramáticas LALR (LookAhead Left Recursive) o LR(1).

Autómata de Pila



- El estado actual es siempre el del tope de la pila.
- Inicialmente, la pila contiene sólo el estado 0 y no se ha leído ningún token.
- El autómata tiene 4 acciones disponibles:
  - **shift**,
  - **reduce**,
  - **accept**,
  - y **error**.

Autómata de Pila



- De acuerdo con el estado actual, el parser decide si necesita un token para decidir la próxima acción.
- Si necesita un token y no lo tiene, llama a yylex para obtener el próximo token.
- La acción puede provocar que se apilen y desapilen estados en la pila, y que el token sea procesado o no.

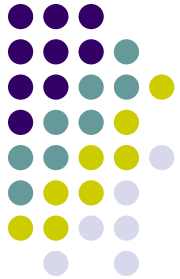
Autómata de Pila

- Una acción ***shift***, implica la existencia de un token leído.
- Ejemplo: En un estado, puede haber una acción:

  IF shift 34

  - Si el token leído es IF, el estado 34 se convertirá en el estado actual (en el tope de la pila).
  - Se pide un nuevo token.

Autómata de Pila



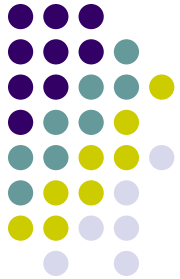
- Una acción ***reduce*** evita que la pila crezca sin límites.
- Se ejecuta cuando el parser ha visto el lado derecho de una regla y está listo para reemplazar el lado derecho por el izquierdo.
- Las acciones *reduce* son asociadas con reglas individuales de la gramática.

  La acción:

  . reduce 18

  se refiere a la regla 18 de la gramática.
- Hace que se desapilen tantos estados como símbolos tenga la regla del lado derecho.

Autómata de Pila

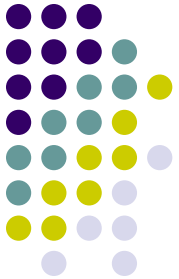


- Al desapilar estados, luego de un ***reduce***, queda descubierto el estado en que el parser estaba al comenzar a procesar la regla.
- Con este estado y el símbolo del lado izquierdo de la regla, se ejecuta un shift de un nuevo estado a la pila, pero sin leer un nuevo token.
- Esta acción se llama ***goto***.
- Ejemplo:

  A goto 20

  A es el lado izquierdo de la regla después del reduce, y 20 será el nuevo estado actual.

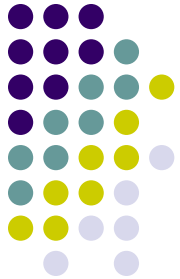
Autómata de Pila



- La acción ***accept*** indica que el parser ha visto la entrada completa y que ésta cumple con la especificación.

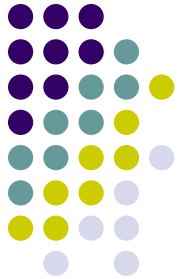
  Esta acción aparece sólo cuando el token leído es la marca de fin de archivo.
- La acción ***error*** representa un lugar donde el parser no puede continuar el proceso de acuerdo con la especificación.

Ejemplo: Especificación YACC



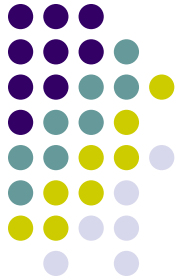
```
%token   A   B   C
%%
lista       :       inicio   fin
            ;
inicio      :       A   B
            ;
fin         :       C
```

Ejemplo: Autómata



```
state 0
        $accept  :  _lista  $end

        A  shift  3
        .  error

        lista  goto  1
        inicio  goto  2

state 1
        $accept  :  lista_$end

        $end  accept
        .  error

state 2
        lista  :  inicio_fin

        C  shift  5
        .  error

        fin  goto  4

state 3
        inicio  :  A_B

        B  shift  6
        .  error

state 4
        lista  :  inicio  fin_      (1)

        .  reduce  1

state 5
        fin  :  C_                  (3)

        .  reduce 3

state 6
        inicio  :  A  B_            (2)

        .  reduce  2
```

Conflictos en gramáticas Yacc



- Ante la regla

  expr : expr ´-´ expr

- La entrada:

  expr – expr – expr

  Puede ser reconocida como:

  (expr – expr) – expr (Asociatividad a izquierda)

  reduce

  O

  expr – (expr – expr) (Asociatividad a derecha)

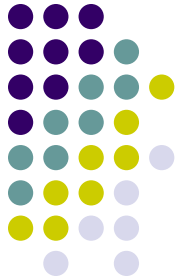
  shift

Conflictos en gramáticas Yacc

- En un conflicto **shift-reduce**, la acción por defecto es el shift

- En un conflicto **reduce-reduce**, el defecto es reducir por la primera regla (en la especificación Yacc)

Conflictos en gramáticas Yacc



- Para la regla:

```
sent  :  IF  ´(´  cond  ´)´  sent
      |  IF  ´(´  cond  ´)´  sent  ELSE  sent
      ;
```

- Con la entrada:

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

- El parser puede reconocer:

```
IF  (  C1  )
{
    IF  (  C2  )
        S1
}
ELSE
    S2
```

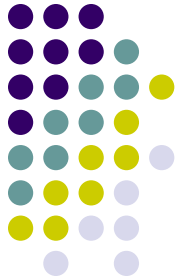
(Reduce)

o

```
IF  (  C1  )
{
    IF  (  C2  )
        S1
    ELSE
        S2
}
```

(Shift)

Conflictos en gramáticas Yacc



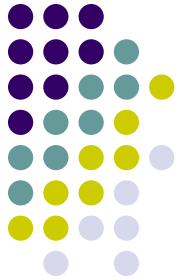
- El ejemplo anterior corresponde a un conflicto shift-reduce.
- En el autómata, se verá como:

```
23:  shift-reduce conflict (shift 45, reduce 18) on ELSE
 state 23
    sent   :   IF   (   cond   )   sent_          (18)
    sent   :   IF   (   cond   )   sent_ELSE   sent
    ELSE   shift  45
    .          reduce  18
```

Precedencia

```
%left  ´+´  ´-´
%left  ´*´  ´/´

%%

expr :  expr  ´=´  expr
     |  expr  ´+´  expr
     |  expr  ´-´  expr
     |  expr  ´*´  expr
     |  expr  ´/´  expr
     |  ´-´  expr       %prec  ´*´
     |  ID
     ;
```

Reglas de desambiguación



- Se consideran las precedencias y asociatividades para aquellos tokens y literales que las tengan.
- Cuando hay un conflicto **reduce-reduce** o un conflicto **shift-reduce** y, ni el símbolo de entrada ni la regla tienen precedencia y asociatividad, entonces se usan las dos reglas de desambiguación descriptas anteriormente, y los conflictos son reportados.
- Si hay un conflicto **shift-reduce**, y tanto la regla de la gramática como el carácter de entrada tienen precedencia y asociatividad asociadas con ellos, el conflicto se resuelve en favor de la acción (**shift** o **reduce**) asociada con la precedencia más alta. Si las precedencias son iguales, se usa la asociatividad.
  - La asociatividad a izquierda implica **reduce**;
  - La asociatividad a derecha implica **shift**;
  - La no asociatividad implica **error**.