Seminario Bison

Compiladores

Dpto. de Ingeniería de la Información y las Comunicaciones

FACULTAD DE INFORMÁTICA

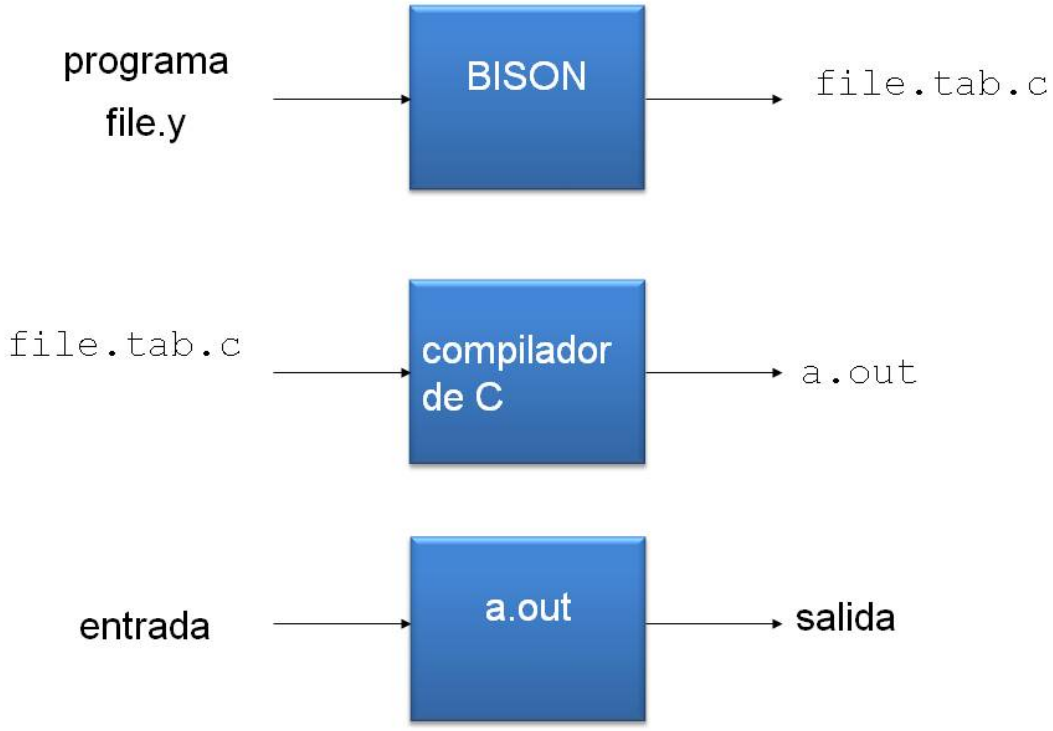
UNIVERSIDAD DE MURCIA

Estudio de la Herramienta Bison

1. Nociones básicas
2. Flex y Bison
3. El entorno Bison
4. Cómo trabaja el parser
5. Ambigüedad y conflictos
6. Precedencia y Asociatividad
7. Manejo de errores
8. Características Avanzadas de Bison

Definición y uso

- Bison es un generador de analizadores sintácticos LALR.
- Bison es compatible con Yacc (otro generador de analizadores sintácticos).
- Construcción de un traductor usando Bison:



Partes de un programa fuente en Bison: Formato y Declaraciones

Formato de un fichero.y :

{declaraciones}
%%
{reglas}
%%
{rutinas de apoyo en C}

- **Declaraciones** :
  - Declaraciones ordinarias en C, delimitadas por %{ y %}
  - Declaraciones de los componentes léxicos de la gramática, del símbolo inicial, etc. (esto se explica más adelante).
  - Otras declaraciones (asociatividad y precedencia,...)

  Podría estar vacía.

Partes de un programa fuente en Bison: Reglas

- **Reglas** : Cada una de ellas consta de una producción de la gramática y la acción semántica asociada. Es decir,

  <lado izquierdo>→ <alt 1>|<alt 2>|... |<alt n>

  pasaría a ser en Bison:

  <lado izquierdo>: <alt 1> {acción semántica 1}
  |<alt 2> {acción semántica 2}
  ...
  |<alt n> {acción semántica n}
  ;

  → Un carácter simple entre comillas 'c' se considera como el **símbolo terminal** c.

  → Las cadenas sin comillas de letras y dígitos no declaradas como componentes léxicos se consideran **no terminales**.

  → El primer lado izquierdo se considera como **símbolo inicial** por defecto, o bien se declara: *%start símbolo*.

Partes de un programa fuente en Bison: Reglas

- → Pueden definirse también otras variables para ser usadas por las acciones.
- → Tanto las declaraciones como las definiciones deben aparecer en la sección de declaraciones, entre %{ y %}. Tienen ámbito global.
- → Deben evitarse los nombres de variables que empiecen por yy, pues todos los nombres de variables internas de Bison comienzan de esta forma.

Partes de un programa fuente en Bison: Reglas Acciones Semánticas

- → Una **acción semántica** es una secuencia de proposiciones en C, dónde $$ se refiere al valor del atributo asociado con el no terminal del lado izquierdo, mientras que $i se refiere al valor asociado con el i-ésimo símbolo gramatical del lado derecho. Se ejecuta siempre que se reduzca por la producción asociada. Por defecto es {$$=$1;}.
- → Una acción semántica no tiene por qué venir al final de su regla. Bison permite que una acción sea escrita en mitad de una regla, y por tanto se ejecutará en esa posición. Esta acción devuelve un valor, que queda accesible a las acciones que están a su derecha.

Partes de un programa fuente en Bison:Reglas Acciones Semánticas

Ejemplo Reglas Semánticas

A : B {$$=1;} C {x=$2; y=$3;}

El efecto es poner x a 1 e y al valor devuelto por C.

→ Las acciones en medio de una regla son manejadas por Bison como si fueran un nuevo nombre de no-terminal, con una nueva regla para él con la cadena vacía en su parte derecha y, como acción de esa regla, ella misma.

Partes de un programa fuente en Bison:Reglas Acciones Semánticas

El ejemplo anterior lo manejaría así:

Ejemplo Reglas Semánticas

```
$ACT : /* empty */ {$$=1;};
A : B $ACT C {x=$2; y=$3;};
```

NOTA: Puede haber conflictos cuando ocurre una acción interior en una regla antes de que el parser pueda estar seguro de qué regla está siendo reducida.

→ En muchas aplicaciones, la salida no viene dada directamente con las acciones. En su lugar se construye en memoria una estructura de datos (p.e. un **árbol de análisis**) y se aplican transformaciones en él antes de que la salida sea generada.

Partes de un programa fuente en Bison: Reglas

→ Los árboles de análisis son particularmente fáciles de construir, si se tienen rutinas para realizar y mantener la estructura de árbol deseada.

Ejemplo

Tenemos una función C que se llama nodo, de forma que

```
nodo(L,n1,n2)
```

crea un nodo con etiqueta L y descendientes n1 y n2, y devuelve el índice del nuevo nodo. El árbol de análisis podría ser realizado suministrando acciones en la especificación como:

```
expr: expr '+' expr
      {$$=nodo('+', $1, $3);};
```

Partes de un programa fuente en Bison: Rutinas de apoyo en C

- **Rutinas de apoyo en C** . Se debe proporcionar un analizador léxico, *yylex()*, que produzca componentes léxicos (es decir, los componentes declarados en la primera sección de Bison) con sus valores de atributos asociados, que se comunican al sintáctico a través de la variable *yylval*.
  Además, se pueden añadir otros procedimientos, como rutinas de recuperación de errores.

→ Esta sección también puede estar vacía. Así, la especificación más pequeña legal de Bison es

```
%%
reglas
```

→ Los **blancos**, **tabuladores** y **saltos de línea** son ignorados.

→ Los **comentarios** pueden aparecer entre /* */, como en C.

Partes de un programa fuente en Bison: Rutinas de apoyo en C

→ Los **nombres** pueden ser de longitud arbitraria y pueden estar formados por letras (distinguiendo mayúsculas de minúsculas), puntos, signos de subrayado y dígitos. Los nombres usados en el cuerpo de una regla de gramática pueden representar tokens o símbolos no terminales.

- Los nombres que representan tokens deben ser declarados. Esto se hace de la siguiente forma

  ```
  % token nombre1 nombre2 ....
  ```

  en la sección de declaraciones.

Ejemplo

Construcción de una calculadora sencilla que lea una expresión aritmética, la evalúe, y después imprima su valor numérico. Gramática:
E → E+T |T
T → T*F |F
F → (E) |digito

Ejemplo 1: calculadora.y (sólo reglas)

```
%token DIGITO

%%
entrada : /*vacio*/ { printf("Aplica entrada -> lambda \n"); }
| entrada linea { printf("Aplica entrada -> entrada linea \n"); }
;
linea : expr '\n' { printf("Aplica linea -> expr \n"); }
;
expr : expr '+' term { printf("Aplica expr -> expr + term \n"); }
| term { printf("Aplica expr->term\n"); }
;
term : term '*' fact { printf("Aplica term -> term * fact \n");}
| fact { printf("Aplica term -> fact \n");}
;
fact : '(' expr ')'{ printf("Aplica fact -> (expr) \n"); }
| DIGITO { printf("Aplica fact -> DIGITO \n"); }
;
%%
```

Ejemplo 1: calculadora.y (Semántica)

```
%token   DIGITO

%%
entrada : /*empty*/
| entrada linea
;
linea : expr '\n' {printf("%d\n",$1);}
;
expr : expr '+' termino {$$=$1+$3;}
| termino
;
termino : termino '*' factor {$$=$1*$3;}
| factor
;
factor : '(' expr ')' {$$=$2;}
| DIGITO
;
%%
```

Flex y Bison

- Flex genera una función *yylex()*, que es llamada por el programa principal de Bison por defecto.
- Cada regla Flex debería terminar con

  ```
  return(token);
  ```

  devolviendo el valor apropiado de token.
- La interacción se realiza de la siguiente forma:

  ```
  bison -d y1.y
  flex l1.l
  gcc y1.tab.c lex.yy.c -lfl -o salida
  ```

- También podría compilarse el fichero de salida de flex como parte del fichero de salida Bison, poniendo:

  ```
  #include "lex.yy.c"
  ```

  en la primera sección de Bison (entre %{ y %}) o bien en la última, y omitiendo en el comando de compilación gcc el fichero `lex.yy.c`.

Flex y Bison

- No importa el orden de generación de los programas Flex y Bison.
- Usando la opción *-d* de Bison se genera `fichero.tab.h`, de definición de tokens, que después puede ser incluido en el programa flex, poniendo

```
%{
#include "fichero.tab.h"
%}
```

en la sección de definiciones del fichero de entrada flex.

El entorno Bison

- La función producida por Bison se llama *yyparse()*, y es de tipo entero.
- Cuando es llamada, ella, a su vez, va llamando a la función *yylex()* para obtener los tokens de entrada. Si detecta un error sin posibilidad de recuperación, devuelve un 1; si el analizador léxico devuelve el final de fichero, el parser acepta y devuelve el valor 0
- Para obtener un programa que funcione, es necesario:
  1. **main()** debe ser definido y debe invocar a yyparse(). P.e.:

     ```
     main() {return(yyparse());}
     ```
  2. **yyerror()**, también debe ser definido para imprimir un mensaje de error cuando ocurre un error sintáctico. Su versión más simple puede ser:

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

     s es una cadena que contiene un mensaje de error.
  3. **yylex()** debe ser declarada.

El entorno Bison

3. **yychar** es una variable que contiene el siguiente token.
4. **yydebug** es una variable que normalmente está a 0. Si no es así, el parser dará información de sus acciones, símbolos de entrada leídos, etc. Esto puede servir para depuración.

Ejemplo 1: calculadora.y

```
%{
#include <stdio.h>
int yylex (void);
void yyerror (char const *);
%}

%token   DIGITO

%%
entrada : /*vacio*/ {printf("Aplica entrada -> lambda \n");}
| entrada linea {printf("Aplica entrada -> entrada linea \n");}
;
linea : expr '\n'{printf("Aplica linea -> expr \n");}
;

expr : expr '+' termino {printf("Aplica expr -> expr + termino \n");}
| termino {printf("Aplica expr->termino\n");}
;
termino : termino '*' factor {printf("Aplica termino -> termino * factor \n");}
| factor {printf("Aplica termino -> factor   \n");}
;
factor : '(' expr ')'{printf("Aplica factor-> (expr) \n");}
| DIGITO {printf("Aplica factor -> DIGITO \n");}
;

%%
```

Ejemplo 1: calculadora.y (Cont.)

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

int yylex() {
    int c;
    while ((c = getchar ()) == ' ' || c == '\t');
    if (isdigit(c)) {
        yylval=c-'0';
        return DIGITO;
    }
    return c;
}

int main (void) {
    return yyparse();
}
```

Ejercicio

Calculadora con Flex

1. Compila el fichero calculadora.y y pruébalo con algunas entradas.
2. Partiendo del ejemplo anterior:
   - Crea el código necesario para tener un analizador léxico usando Flex.
   - Tienes que añadir la cabecera generada por Bison al analizador léxico para que puedan reconocer los mismos tokens.
   - Crea el makefile correspondiente.
3. Una vez que lo anterior funcione:
   - Modifica el analizador léxico y sintáctico para añadir los operadores de resta y división.
   - Modifica lo necesario para reconocer números negativos.
   - Modifica las acciones de Bison para que funcione como una calculadora, en lugar de devolver la secuencia de reglas aplicadas.

Cómo trabaja el Parser

- El analizador sintáctico producido por Bison es una **máquina de estados finitos con una pila**. Además, el analizador léxico es capaz de leer y recordar el siguiente token de entrada (token de anticipación o lookahead).
- El **estado actual** es siempre el del tope de la pila. Los estados vienen etiquetados con enteros pequeños; inicialmente, la máquina está en el estado 0, la pila sólo contiene ese estado y aún no se ha leído ningún token de lookahead.

→ La máquina puede realizar cuatro acciones: **desplaza** (*shift*), **reduce** (*reduce*), **acepta** (*accept*) y **error** (*error*).

→ Cada movimiento se realiza como sigue:

- Según el estado actual, el parser decide si necesita un token de anticipación para decidir qué acción elegir; si necesita uno y no lo tiene, llama a *yylex()* para obtener el siguiente token.
- Usando el estado actual y el token lookahead, si se necesita, el analizador decide su siguiente acción, y la lleva a cabo.

Cómo trabaja el Parser

- La acción **desplaza** (*shift*) es la más común que realiza el parser. Para ella se necesita un token de anticipación. Si estamos en el estado i, y la acción en él es

  $$TOK\ shift\ j$$

  significa que, en el estado i, con el token de anticipación *TOK*, el estado j se convierte en el estado actual (pasa a ser el tope de la pila) y el token *TOK* se consume.
- Con la acción reduce puede ser necesario consultar el símbolo de anticipación para decidir si reducir, pero usualmente no. Las acciones reduce están asociadas con reglas de la gramática, representadas también por medio de enteros pequeños.
  Si en el estado en el que estamos pone

  $$reduce\ k$$

Cómo trabaja el Parser

y la regla nº k es, por ejemplo

$$A : x\ y\ z;$$

se sacan de la pila tantos estados como símbolos tenga la parte derecha de la regla (tres en este caso). Después, usando el estado que ha quedado en el tope y el símbolo de la parte izquierda de la regla (A en este caso), obtenemos el nuevo estado, que introducimos en la pila, después de realizar un desplazamiento de A.
Este nuevo estado lo obtenemos a partir de una acción que hay en el estado que quedó al descubierto:

$$A \text{ goto } l$$

que convierte a l en el estado actual

Cómo trabaja el Parser

- La acción *reduce* es también importante en el tratamiento de acciones dadas por el usuario y valores: cuando se reduce con una regla, el código dado con la regla es ejecutado antes de que la pila sea ajustada. Además de la pila que maneja los estados, otra pila, paralela a ésta, contiene los valores devueltos por el analizador léxico y las acciones. Cuando se produce un *shift*, *yylval* se inserta en la pila de valores. Después de volver del código de usuario, se lleva a cabo la reducción. Cuando se realiza un *goto*, *yyval* se copia en la pila. Las pseudovariables \$1, \$2, ... remiten a esta pila.
- La acción **acepta** (*accept*) indica que la entrada ha sido leída y que es correcta.
- La acción **error** indica un lugar dónde el parser no puede continuar de acuerdo con la especificación. El parser produce un error, intenta recuperar la situación y retomar el análisis.
- Invocando Bison con -v, se crea el fichero fichero.output, que incluye una descripción del parser.

Cómo trabaja el Parser:Ejemplo Calculadora

ejem1.y

```
%start e
%token  POR MAS PARIZ PARDE ID

%%

e : e MAS t          {$$=$1+$3;}
  | t;

t : t POR f {$$=$1*$3;}
  | f  ;

f : PARIZ e PARDE   {$$=$2;}
  | ID  ;
```

ejem1.l

```
 %{
#include "y.tab.h"
extern int yylval;
%}

%%
"*"     return(POR);
"+"   return(MAS);
"("   return(PARIZ);
")"   return(PARDE);
[0-9]+ {yylval=atoi(yytext);return(ID);};
"\n" return(0);
```

Cómo trabaja el Parser: Ejemplo Calculadora

y.tab.h (generado por Bison)

```
# define POR 257
# define MAS 258
# define PARIZ 259
# define PARDE 260
# define ID 261
```

Cómo trabaja el Parser: Ejemplo Calculadora

y.output (generado por Bison)

```
state 0
$accept : . e $end
PARIZ shift 4
ID shift 5

e  goto 1
t  goto 2
f  goto 3

state 1
$accept :  e . $end
e :  e . MAS t

$end  accept
MAS  shift 6

state 2
e :  t .
t :  t . POR f

POR  shift 7
$default  reduce 2

state 3
t :  f .

$default  reduce 4
```

Cómo trabaja el Parser: Ejemplo Calculadora

y.output (generado por Bison) (cont.)

```
state 4
f :  PARIZ . e PARDE

PARIZ  shift 4
ID  shift 5

e  goto 8
t  goto 2
f  goto 3

state 5
f :  ID .

$default  reduce 6

state 6
e :  e MAS . t

PARIZ  shift 4
ID  shift 5

t  goto 9
f  goto 3
```

Cómo trabaja el Parser: Ejemplo Calculadora

y.output (generado por Bison) (cont.)

```
state 7
t :  t POR . f

PARIZ  shift 4
ID  shift 5

f  goto 10

state 8
e :  e . MAS t
f :  PARIZ e . PARDE

MAS  shift 6
PARDE  shift 11

state 9
e :  e MAS t .
t :  t . POR f

POR  shift 7
$default  reduce 1
```

Cómo trabaja el Parser: Ejemplo Calculadora

y.output (generado por Bison) (cont.)

```
state 10
t :  t POR f .

$default  reduce 3
state 11
f :  PARIZ e PARDE .

$default  reduce 5

7/127 terminals, 3/600 nonterminals
7/300 grammar rules, 12/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
7/601 working sets used
memory: states,etc. 28/2000, parser 8/4000
8/3001 distinct lookahead sets
4 extra closures
13 shift entries, 1 exceptions
6 goto entries
3 entries saved by goto default
Optimizer space used: input 37/2000, output 16/4000
16 table entries, 4 zero
maximum spread: 261, maximum offset: 259
```

Ambigüedad y conflictos

- Bison detecta las ambigüedades cuando intenta construir el parser.
- Cuando hay conflictos *shift/reduce* o *reduce/reduce*, se genera el parser de todas formas. Lo hace seleccionando uno de los pasos válidos cuando tenga que elegir, y para decidir qué elección hacer, utiliza las **reglas para deshacer la ambigüedad**. Por defecto, estas reglas son dos:
  1. En un conflicto *shift/reduce*, hace un desplazamiento.
  2. En un conflicto *reduce/reduce*, reduce con la regla de la gramática que aparezca primero en la especificación de Bison (con esto se le da al usuario un control tosco del comportamiento del parser).
- Los **conflictos** pueden producirse por errores en la entrada o lógicos, o porque las reglas de la gramática, aunque consistentes, requieren un parser más complejo que el que Bison puede construir. Además pueden venir provocados por el uso de acciones dentro de las reglas, cuando deben ejecutarse antes de que el parser sepa qué regla aplicar.

Ambigüedad y conflictos

- En estos casos, la aplicación de reglas como las anteriores es inapropiada y conduce a analizadores incorrectos. En general, si al aplicarlas se llega a un analizador correcto, es posible reescribir la gramática sin conflictos.
- Bison siempre informa sobre el número de conflictos *shift/reduce* y *reduce/reduce* resueltos con las dos reglas anteriores.
  Para obtener más información sobre los conflictos encontrados en las reglas por Bison, podemos analizar el fichero *fichero.output*, ejecutando Bison con la opción -v.

Ejemplo

```
stat: IF '(' cond ')' stat
                |IF '(' cond ')' stat ELSE stat
                ;
```

Ambigüedad y conflictos

En y.output tendríamos:

```
8: shift/reduce conflict (shift 9, reduce 1) on ELSE
state 8

stat : IF ( cond ) stat .
stat : IF ( cond ) stat . ELSE stat

ELSE shift 9
$default reduce 1
```

Con el ELSE hace un desplazamiento. En el estado 45 habrá:

```
stat : IF ( cond ) stat ELSE . stat
```

Si el símbolo de entrada no es ELSE, se reduce con la regla de la gramática 18:

```
stat : IF '(' cond ')' stat
```

En este caso, la acción por defecto (shift) es la apropiada. En general, tenemos que comprobarlo.

Ambigüedad y conflictos: Ejemplos

ejem2.l

```
%{
#include "y.tab.h"
%}

%%
if          return(IF);
p           return(P);
a           return(A);
"("         return(PARIZ);
")"         return(PARDE);
else        return(ELSE);
```

ejem2.y

```
%{
%}

%start s

%token  IF P A PARIZ PARDE ELSE

%%

s : IF PARIZ c PARDE s
  | IF PARIZ c PARDE s ELSE s
  | A
  ;

c : P
  ;
```

Ambigüedad y conflictos: Ejemplos

y.output

```
state 0
$accept : . s $end

IF  shift 2
A  shift 3

s  goto 1

state 1
$accept :  s . $end

$end  accept

state 2
s :  IF . PARIZ c PARDE s
s :  IF . PARIZ c PARDE s ELSE s

PARIZ  shift 4

state 3
s :  A .

$default  reduce 3
```

Ambigüedad y conflictos: Ejemplos

y.output (cont.)

```
state 4
s :  IF PARIZ . c PARDE s
s :  IF PARIZ . c PARDE s ELSE s

P  shift 6

c  goto 5

state 5
s :  IF PARIZ c . PARDE s
s :  IF PARIZ c . PARDE s ELSE s

PARDE  shift 7

state 6
c :  P .

$default  reduce 4

state 7
s :  IF PARIZ c PARDE . s
s :  IF PARIZ c PARDE . s ELSE s

IF  shift 2
A  shift 3

s  goto 8
```

Ambigüedad y conflictos: Ejemplos

y.output (cont.)

```
8: shift/reduce conflict (shift 9, reduce 1) on ELSE
state 8
s :  IF PARIZ c PARDE s .
s :  IF PARIZ c PARDE s . ELSE s

ELSE  shift 9
$default  reduce 1

state 9
s :  IF PARIZ c PARDE s ELSE . s

IF  shift 2
A  shift 3

s  goto 10

state 10
s :  IF PARIZ c PARDE s ELSE s .

$default  reduce 2
```

Ambigüedad y conflictos: Ejemplos

y.output (cont.)

```
8/127 terminals, 2/600 nonterminals
5/300 grammar rules, 11/1000 states
1 shift/reduce, 0 reduce/reduce conflicts reported
5/601 working sets used
memory: states,etc. 26/2000, parser 3/4000
6/3001 distinct lookahead sets
5 extra closures
10 shift entries, 1 exceptions
4 goto entries
0 entries saved by goto default
Optimizer space used: input 28/2000, output 17/4000
17 table entries, 7 zero
maximum spread: 262, maximum offset: 262
```

Precedencia y Asociatividad

- En el análisis de expresiones aritméticas, las reglas anteriores para resolver conflictos no son suficientes.
- La mayoría de las construcciones aritméticas usadas normalmente pueden describirse de forma natural mediante la noción de **niveles de precedencia** para operadores, junto con la información sobre **asociatividad izquierda o derecha**.
- Gramáticas ambiguas con reglas apropiadas para quitar la ambigüedad pueden ser usadas para crear analizadores que son más rápidos y más fáciles de escribir que los construidos para gramáticas no ambiguas.
- La noción básica es escribir la gramática con reglas de la forma:

  expr : expr OP expr

  y

  expr : UNARY expr

  para todos los operadores binarios y unarios deseados.

Precedencia y Asociatividad

- Esto crea una gramática muy ambigua, con muchos conflictos. Como reglas para quitar la ambigüedad, el usuario especifica la precedencia de todos los operadores y la asociatividad de los operadores binarios. Esto es suficiente para permitir que Bison genere un parser sin conflictos de acuerdo con esas reglas.
- Las precedencias y asociatividades se ligan a los tokens en la sección de declaraciones.
- Esto se hace con una serie de líneas que comienzan con una palabra clave de Bison: *%left, %right, %nonassoc*, seguidas de una lista de tokens. Los tokens en la misma línea se asume que tienen la misma precedencia y asociatividad; las líneas son listadas en orden de precedencia creciente.
  - *%left* describe asociatividad por la izquierda.
  - *%right* describe asociatividad por la derecha.
  - *%nonassoc* describe operadores que no pueden asociarse con ellos mismos. (Ej.: .LT. de FORTRAN, es decir, A .LT. B .LT. C es ilegal).

Precedencia y Asociatividad

- → Un token declarado con *%left*, *%right* y *%nonassoc* no necesita, pero puede, ser declarado también con *%token*.
- → A veces un operador unario y un operador binario tienen la misma representación simbólica pero distintas precedencias (**ej.**: - como unario tiene más precedencia que como binario).
  %prec cambia el nivel de precedencia asociado con una regla particular de la gramática.
  Aparece inmediatamente después que el cuerpo de la regla, antes de la acción o del punto y coma, y es seguido por un token.

Precedencia y Asociatividad

Ejemplo

```
%right '='
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
expr : expr '=' expr
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '-' expr %prec UMINUS
| NAME
;
```

- → '+' y '-' tienen asociatividad por la izquierda y menor precedencia que '*' y '/', que también son asociativos por la izquierda.
- → '=' es asociativo por la derecha y tiene la menor precedencia.
- → La entrada a=b=c*d-e-f*g se estructuraría como a=(b=(((c*d)-e)-(f*g))).
- → Por último, el operador unario '-' tiene la máxima precedencia.

Precedencia y Asociatividad

- **Reglas para quitar la ambigüedad** usando precedencias y asociatividades para resolver conflictos:
  - Las precedencias y asociatividades están asociadas a los tokens y literales para los que se indica expresamente.
  - A cada regla de la gramática se le asocia una precedencia y una asociatividad: las del último token o literal en el cuerpo de la regla. Si se usa `%prec`, la opción por defecto se anula.
    Si el último token (o el referenciado por `%prec`) no tiene definida precedencia o asociatividad, la regla tampoco.
  - Cuando se produce un conflicto *reduce/reduce* o *shift/reduce* y el símbolo de entrada o la regla gramatical no tienen precedencia y asociatividad, se usan las dos reglas por defecto (ver transparencia 32).

Precedencia y Asociatividad

- Si en un conflicto *shift/reduce* tanto el símbolo de entrada como la regla de producción tienen precedencia y asociatividad, el conflicto se resuelve a favor de la acción (shift o reduce) asociada con la más alta precedencia (reducción si la regla tiene mayor precedencia y desplazamiento si el símbolo de entrada tiene más precedencia).
  Si tienen la misma precedencia, se usa la asociatividad; si es por la izquierda se reduce, por la derecha se desplaza, y la no asociatividad implica error.

Los conflictos resueltos por precedencia y asociatividad no son tenidos en cuenta en el informe de conflictos de Bison, y por tanto pueden ocultar errores. Hay que tener cuidado. **Es bueno utilizar y.output.**

Manejo de errores

No es aceptable parar el procesamiento cuando se encuentra un error, sino que se debe seguir escaneando la entrada para encontrar más errores sintácticos. Necesitamos volver a empezar a analizar después de un error. Los algoritmos que hacen esto suelen descartar tokens de la cadena de entrada, e intentar ajustar el analizador para que pueda continuar con la entrada.

→ Para permitir al usuario algún control sobre este proceso, Bison ofrece un mecanismo general:

- Se reserva para el manejo de errores el token de nombre *error*. Puede ser usado en reglas gramaticales, cuando se espera un error y puede tener lugar una recuperación.
- Se decide qué no-terminales 'principales' tendrán recuperación de errores asociados a ellos (expresiones, proposiciones, bloques, procedimientos,...).
- Para cada uno de ellos se añade la producción A $\rightarrow$ error α, siendo α una cadena de símbolos gramaticales, a veces vacía.
- Bison genera el analizador como si esas producciones fueran normales, pero cuando este analizador encuentra un error, trata a los estados cuyos conjuntos de elementos contengan producciones de error, de manera especial.

Manejo de errores

- Al encontrar un error, Bison extrae símbolos de la pila hasta que encuentra un estado dónde el token *error* es legal, es decir, en él hay algo como:

$$A \rightarrow . \text{ error } \alpha$$

  Entonces desplaza un token ficticio *error* a la pila, como si lo hubiera encontrado en la entrada.

  - Si α es λ, se produce inmediatamente una reducción a A, se invoca la acción semántica asociada a la producción A $\rightarrow$ error (rutinas de recuperación de errores especificadas por el usuario, para intentar reinicializar tablas, etc.), y se eliminan símbolos de la entrada hasta que se encuentra uno con el que poder seguir el análisis normal. **Ej.**: `stat : error`.
  - Si α no es vacía, salta la entrada buscando una subcadena que se pueda reducir a α. Si α sólo consta de terminales, entonces busca esta subcadena en la entrada y la desplaza a la pila. Reduce *error*α a A, y prosigue el análisis. **Ej.**: *stat : error ';'.*

Manejo de errores

- Para **prevenir una cascada de errores**, el analizador, después de detectar un error, permanece en estado de error hasta que han sido leídos y deplazados con éxito tres tokens. Si se detecta un error cuando está en estado de error, no se da mensaje y se borra el token.

→ Si no han sido especificadas reglas de error especiales, el procesamiento para cuando se detecta un error.

→ Otra forma de regla de error se presenta en aplicaciones interactivas, dónde puede ser deseable permitir que vuelva a introducirse una línea después de un error. Ejemplo:

```
input: error '\n' {printf("vuelva a introducir
                    línea:");} input {$$=$4;}
```

Manejo de errores

- *Problema que puede surgir*: Puesto que el parser debe procesar correctamente tres tokens de entrada antes de admitir que tiene resincronizada correctamente la entrada después del error, puede ocurrir que en la línea que ha vuelto a introducirse se produzca un error en los dos primeros tokens. En este caso borraría los tokens malos y no daría mensaje. Esto es inaceptable.
  Existe un mecanismo para **forzar al analizador a creer que un error ha sido recuperado ya**:

  ```
  yyerrok;
  ```

  que vuelve al parser a un modo normal.

→ Después de producirse un error, el siguiente token es aquel en que el error fue descubierto. A veces esto es inapropiado, por ejemplo, puede que una acción de recuperación de errores se encargue de encontrar el lugar donde reanudar la entrada. En este caso, el token de anticipación previo debe ser borrado. Esto puede hacerse por medio de

  ```
  yyclearin;
  ```

Manejo de errores

Ejemplo

```
stat : error
       {resincronizar();
       yyerrok;
       yyclearin;};
```

Características Avanzadas de Bison

- **Simulación de Error y Accept en las acciones**
  - *YYACCEPT* es una macro que hace que Bison devuelva el valor 0 (para informar de éxito).
  - *YYERROR* hace que el parser se comporte como si el actual símbolo de entrada hubiera sido erróneo. Con esta macro se llama a yyerror y tiene lugar una recuperación de errores.

  Pueden ser usadas para simular analizadores sintácticos con múltiples marcadores de final de fichero o chequeo de sintaxis sensible al contexto.
- **Acceso a valores desde las reglas**
  - Una acción puede referirse a valores devueltos por acciones a la izquierda de la regla actual. El mecanismo es el mismo que con acciones ordinarias, $i, pero en este caso i puede ser 0 o negativo.

Características Avanzadas de Bison

Ej.- Ejemplo

```
sent : adj noun verb adj noun {...}
adj : THE {$$=THE;}
    | YOUNG {$$=YOUNG;}
        ...
    ;
noun : DOG {$$=DOG;}
    | CRONE {(*)if($0==YOUNG)
                    {printf("what?\n");}
                    $$=CRONE;};
    ...
```

(*) Chequea que el token precedente desplazado no fuera YOUNG.
Hace falta conocer muy bien qué puede haber antes de noun.

Características Avanzadas de Bison

Soporte para tipos de valores arbitrarios

- Por defecto, los valores devueltos por las acciones y el analizador léxico son enteros.
- Bison también puede soportar valores de otros tipos, incluyendo estructuras.
- La pila de valores de Bison se declara como una unión de los tipos de los valores deseados. El usuario declara la unión, y asocia los nombres de miembros de la unión a cada token y a cada no terminal que tenga un valor.
- Cuando el valor es referenciado con $$ o $n, Bison insertará automáticamente el tipo apropiado, de forma que no tendrán lugar conversiones inesperadas.

Características Avanzadas de Bison

- Existen tres mecanismos para proporcionar esto:
  1. *Definición de la unión.* Debe hacerlo el usuario, puesto que otros programas como el analizador léxico deben conocer los nombres de los miembros de la unión. Puede hacerse de dos formas:
     - (a) Se pone en la sección de declaraciones

       `%union {cuerpo de la unión...}`

       Declara la pila de valores Bison y las variables externas *yylval* e *yyval*, con tipos iguales a esta unión.
       → Si se ejecuta Bison con la opción -d la declaración de la unión se copia en *y.tab.h*.
     - (b) Se puede declarar la unión en un fichero cabecera, usando un *typedef* para definir la variable *YYSTYPE*:
       `typedef union {cuerpo de la union...}`
       `YYSTYPE;`
       Después se incluiría este fichero en la sección de declaraciones con %{ y %}.

Características Avanzadas de Bison

2. Forma de asociar un nombre de miembro de una unión con tokens y no terminales. La construcción <*nombre*> se usa para indicar el nombre de un miembro de una unión.
   a. Para asociar el tipo con los tokens, se pone <nombre> entre %token, %left, %right o %nonassoc y la lista de tokens listados.
   b. Para asociar el tipo con los no terminales, se pone <nombre> entre %type y la lista de no terminales.

Ej.-

```
%left <optype>'+''-'
%type <nodetype>expr stat
```

Características Avanzadas de Bison

3. Mecanismo para describir el tipo de aquellos pocos valores dónde Bison no puede determinarlo fácilmente.
   Por ejemplo, cuando hay una acción dentro de una regla, esta acción no tiene tipo a priori, o cuando se referencian valores del contexto de la izquierda. En estos casos, un tipo puede ser impuesto en una referencia, insertando <nombre > inmediatamente después del primer $

Ej.-

```
rule : aaa {$<intval>$=3;} bbb
    {fun($<intval>2,$<other>0);}
```

Ejercicio

Ejemplo 2: Calculadora con variables

Usar los archivos `calc.l` y `calc.y` (del Aula Virtual) para realizar una calculadora que evalúe las expresiones aritméticas previas a cada ';', de manera que, ante una entrada como:

```
8*5+8;
(9-4)*27;
99///23;varx=8*100+1000;
4+3;16/2*0;
```

produzca una salida semejante a la siguiente:

```
Expr. 1: 48
Expr. 2: 135
Expr. 3: Se ha producido un error en esta expresion
Expr. 4: La variable varx toma el valor 1800
Expr. 5: 7
Expr. 6: 0
```

Ejercicio

Calculadora con variables

Para esto, hay que realizar los siguientes pasos:

1. Añadir las reglas de precedencia y asociatividad necesarias para eliminar la ambigüedad de la gramática.
2. Añadir las acciones semánticas necesarias para cada regla.
3. Declarar los tipos de los atributos asociados a cada terminal y no terminal en caso de necesitarlos.