Bison

Álvar Arnaiz González

2022

Departamento de Ingeniería Informática
Universidad de Burgos

Tabla de contenidos

1. Bison

¿Qué es?

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical, para una gramática independiente del contexto LALR(1), en un programa en C que analiza dicha gramática.

El analizador creado por bison lee una secuencia de componentes léxicos como entrada, y los agrupa utilizando las reglas gramaticales. Si la entrada es válida, el resultado final es que la secuencia de componentes léxicos completa se reduce al axioma de la gramática.

Normalmente bison se usa junto con flex: el analizador generado por flex es el encargado de realizar el análisis léxico, el cual pasa los tókenes al analizador sintáctico generado por bison.

Código fuente bison

El fuente bison sigue una estructura similar a la de flex, los «%%» definen los cambios de sección. Deben aparecer al inicio de la línea (sin espacios o tabuladores por delante, es decir, en la columna 1).

```
// sección de definiciones
%%
// sección de reglas/producciones
%%
// rutinas de usuario
```

Código fuente bison

Sección de definiciones

En la sección de definiciones pueden aparecer:

1. Código fuente C: definiciones de variables, cabeceras de funciones...
2. Definiciones de tókenes.
3. Directivas para bison (se verán más adelante).

Código fuente bison

Sección de definiciones

En la sección de definiciones pueden aparecer:

1. Código fuente C: definiciones de variables, cabeceras de funciones...
2. Definiciones de tókenes.
3. Directivas para bison (se verán más adelante).

%%

Código fuente bison

Sección de definiciones

En la sección de definiciones pueden aparecer:

1. Código fuente C: definiciones de variables, cabeceras de funciones...
2. Definiciones de tókenes.
3. Directivas para bison (se verán más adelante).

%%

Sección de reglas

Se especifican las producciones de la gramática a analizar y las acciones semánticas asociadas.

Código fuente bison

Sección de definiciones

En la sección de definiciones pueden aparecer:

1. Código fuente C: definiciones de variables, cabeceras de funciones...
2. Definiciones de tókenes.
3. Directivas para bison (se verán más adelante).

%%

Sección de reglas

Se especifican las producciones de la gramática a analizar y las acciones semánticas asociadas.

%%

Código fuente bison

Sección de definiciones

En la sección de definiciones pueden aparecer:

1. Código fuente C: definiciones de variables, cabeceras de funciones...
2. Definiciones de tókenes.
3. Directivas para bison (se verán más adelante).

%%

Sección de reglas

Se especifican las producciones de la gramática a analizar y las acciones semánticas asociadas.

%%

Rutinas de usuario

Se definen las funciones/rutinas de usuario. Ejemplo: `main`, `yyerror`...

Compilación y ejecución

Como se ha indicado, bison se suele utilizar en combinación con flex. De este modo, los pasos para generar el analizador sintáctico son:

1. Generar el analizador léxico a partir del código flex (extensión «`.l`»), ejecutando:
   `flex analizador_lexico.l`
2. Generar el analizador sintáctico a partir del código bison (extensión «`.y`»), ejecutando[1]:
   `bison -yd analizador_sintactico.y`
3. Compilar el fuente en C generado por flex y bison, ejecutando:
   `gcc lex.yy.c y.tab.c -o nombre_ejecutable -lfl`
4. Ejecutar el programa[2]: `./nombre_ejecutable`

[1] Bison genera un fichero «`y.tab.h`» para poder comunicar ambos analizadores. Este nombre se debe a haber ejecutado `bison` con `-y` (modo compatibilidad con yacc).

[2] En Linux, si fuese en Windows se habrá generado un «`.exe`».

Análisis shift-reduce

Cuando bison procesa una gramática, crea un conjunto de estados que reflejan las posibles posiciones en una o más de las reglas parcialmente procesadas.

Mientras lee tókenes que no completan una regla, los pone en una pila (*shift*) y cambia a un nuevo estado que refleja que se acaba de leer el token.

Cuando encuentra todos los símbolos que completan el consecuente de una producción, elimina todos sus símbolos de la pila y pone el antecedente correspondiente (*reduce*) en el tope de pila. Cambiando igualmente el estado al que corresponda de acuerdo con las tablas de análisis sintáctico LALR(1).

Cuando se reduce una regla, bison ejecuta el código asociado con esa regla: esta es la forma de «manipular» el «material» que se está analizando.

2. Definición de la gramática

Símbolos terminales y no terminales

Un símbolo no terminal en la gramática formal se representa en bison como un identificador, similar a un identificador en C. Por convención, debería estar en minúsculas. Ejemplos: `axioma`, `expr` o `stmt`.

La representación en bison para un símbolo terminal se llama también un tipo de token. Los tipos de tókenes también se pueden representar como identificadores al estilo de C. Por convención, estos identificadores deberían estar en mayúsculas para distinguirlos de los no terminales. Ejemplos: `INTEGER`, `IDENTIFICADOR`, `IF` o `RETURN`.

Los caracteres simples entre comillas se consideran como componentes léxicos[3]: por ejemplo, «`'+'`», «`'-'`» o «`'*'`».

El símbolo terminal «`error`» se reserva para la recuperación de errores.

[3] Siempre y cuando se devuelva desde el analizador léxico (flex) con una regla como la siguiente: «`. return yytext[0];`».

Símbolos terminales y no terminales: ejemplo

Ejemplo de una gramática para realizar asignaciones de enteros a variables.

- Analizador léxico: flex

```
%{
#include "y.tab.h"
%}
%%
[0-9]+    return NUM;
[a-z]+    return ID;
":="      return IGUAL;
%%
```

- Analizador sintáctico: bison

```
%token NUM ID IGUAL
%%

axioma: asigna ;

asigna: ID IGUAL NUM ;
```

Como se puede ver, los tókenes definidos en bison deben ser devueltos por flex mediante `return`.

3. Metavariables

Intercambio de información: flex → bison

Generalmente es necesario intercambiar información entre el analizador léxico y sintáctico. Bison genera un fichero «y.tab.h» con una variable para este fin: `yylval`.

Bison puede acceder a dicha información mediante las metavariables: «$1», «$2»…«$n».

Partiendo del ejemplo anterior, suponer que se desea enviar el número reconocido por el analizador léxico.

- Analizador léxico: flex

```
 #include "y.tab.h"
%%
[0-9]+  {yylval=atoi(yytext);
         return NUM;}
[a-z]+   return ID;
":="     return ASIG;
%%
```

- Analizador sintáctico: bison

```
%token NUM ID ASIG
%%

axioma: asigna ;


asigna: ID ASIG NUM {printf("Núm: %d", $3);} ;
```

Intercambio de información: metavariables

Además, suele ser necesario intercambiar información entre diversas producciones

Esto se realiza en bison mediante las metavariables: «$$», «$1», «$2»…«$n».

Partiendo del ejemplo anterior, suponer que se desea imprimir el número al reconocer el axioma (y no en la producción `asigna` como en el caso anterior).

- Analizador léxico: flex

```
 #include "y.tab.h"
%%
[0-9]+  {yylval=atoi(yytext);
         return NUM;}
[a-z]+   return ID;
":="     return ASIG;
%%
```

- Analizador sintáctico: bison

```
%token NUM ID ASIG
%%

axioma: asigna {printf("Núm: %d", $1);} ;

asigna: ID ASIG NUM {$$ = $3;} ;
```

Como se puede ver, «$$» representa información que se desea dejar en el tope de pila.

Metavariables y la pila

Para entender cómo se numeran las metavariables y cómo se realiza el intercambio de información, hay que saber que bison dispone de una pila (no confundir con la pila de estados de flex).

Como se ha explicado, bison realiza desplazamientos y reducciones en función de los caracteres leídos y de la gramática codificada.

- Mientras lee tókenes que no completan una regla, los deplaza a la pila.
- Cuando en la pila encuentra todos los símbolos que completan el consecuente de una producción, elimina todos sus símbolos de la pila y pone el antecedente de la producción en el tope de pila.

Por este motivo, en una producción como la siguiente las metavariables empiezan en 1 y así van hasta el final del consecuente (las acciones semánticas también ocupan posición, como se verá más adelante):

```
        $1  $2  $3
asigna: ID ASIG NUM ;
```

Metavariables: ejemplo de calculadora

La siguiente implementación realiza operaciones aritméticas e imprime el resultado.

- Analizador léxico: «eje1.l»

```
%{
#include "y.tab.h"
%}
%%
[0-9]+  { yylval=atoi(yytext);
          return NUM;
        }

.       return yytext[0];

%%
```

- Analizador sintáctico: «eje1.y»

```
%token NUM
%%
axioma: exp         { printf("=%d", $1); };
exp: exp '+' term   { $$=$1+$3; }
   | exp '-' term   { $$=$1-$3; }
   | term           { $$=$1; }
   ;
term: term '*' fac  { $$=$1*$3; }
    | fac           { $$=$1; }
    ;
fac: NUM            { $$=$1; }
   | '(' exp ')'    { $$=$2; }
   ;
%%
```

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

1. Bison inicia el análisis pidiendo un token a flex.

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

1. Bison inicia el análisis pidiendo un token a flex.
2. Flex analizará la entrada hasta encontrar un token: convertirá el texto a número con la función `atoi`, dejará el número en `yylval` y devolverá el control a bison indicándole que ha encontrado un `NUM`.

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

1. Bison inicia el análisis pidiendo un token a flex.
2. Flex analizará la entrada hasta encontrar un token: convertirá el texto a número con la función `atoi`, dejará el número en `yylval` y devolverá el control a bison indicándole que ha encontrado un `NUM`.
3. Bison reconoce la producción `fac -> NUM`. Realiza la acción asociada: colocar en el tope de pila el número asociado al token. Reduce por dicha producción.

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

1. Bison inicia el análisis pidiendo un token a flex.
2. Flex analizará la entrada hasta encontrar un token: convertirá el texto a número con la función `atoi`, dejará el número en `yylval` y devolverá el control a bison indicándole que ha encontrado un `NUM`.
3. Bison reconoce la producción `fac -> NUM`. Realiza la acción asociada: colocar en el tope de pila el número asociado al token. Reduce por dicha producción.
4. A partir de ahí el control irá pasando de flex a bison a medida que se reconozcan tókenes hasta que no queden más por leer. Observa que las producciones de las operaciones aritméticas dejan en el tope de pila el resultado de dicha operación (ejemplo: `$$=$1+$3;` deja en el tope de pila la suma de los dos números).

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

1. Bison inicia el análisis pidiendo un token a flex.
2. Flex analizará la entrada hasta encontrar un token: convertirá el texto a número con la función `atoi`, dejará el número en `yylval` y devolverá el control a bison indicándole que ha encontrado un `NUM`.
3. Bison reconoce la producción `fac -> NUM`. Realiza la acción asociada: colocar en el tope de pila el número asociado al token. Reduce por dicha producción.
4. A partir de ahí el control irá pasando de flex a bison a medida que se reconozcan tókenes hasta que no queden más por leer. Observa que las producciones de las operaciones aritméticas dejan en el tope de pila el resultado de dicha operación (ejemplo: `$$=$1+$3;` deja en el tope de pila la suma de los dos números).
5. Tras los desplazamientos/reducciones correspondientes, si la entrada estaba bien formada (cumplía con la gramática), bison llegará a tener `exp` en el tope de pila.

Metavariables: ejemplo de calculadora (paso a paso)

¿Cómo funciona dicho ejemplo?

1. Bison inicia el análisis pidiendo un token a flex.
2. Flex analizará la entrada hasta encontrar un token: convertirá el texto a número con la función `atoi`, dejará el número en `yylval` y devolverá el control a bison indicándole que ha encontrado un `NUM`.
3. Bison reconoce la producción `fac -> NUM`. Realiza la acción asociada: colocar en el tope de pila el número asociado al token. Reduce por dicha producción.
4. A partir de ahí el control irá pasando de flex a bison a medida que se reconozcan tókenes hasta que no queden más por leer. Observa que las producciones de las operaciones aritméticas dejan en el tope de pila el resultado de dicha operación (ejemplo: `$$=$1+$3;` deja en el tope de pila la suma de los dos números).
5. Tras los desplazamientos/reducciones correspondientes, si la entrada estaba bien formada (cumplía con la gramática), bison llegará a tener `exp` en el tope de pila.
6. Finalmente, realizará la acción asociada (imprimir el resultado): reducirá por el axioma y finalizará el análisis.

Metavariables: ejemplo par e impar

En ocasiones puede ser interesante acceder a información que otras producciones previas han dejado en la pila.

Esto se realiza en bison mediante las metavariables: «$0», «$-1», «$-2»…«$-n».

Supongamos la siguiente gramática:

- Analizador léxico: flex

```
 #include "y.tab.h"
%%
[0-9]*[02468]   {yylval=atoi(yytext);
                 return NPAR;}
[0-9]*[13579]   {yylval=atoi(yytext);
                 return NIMPAR;}
%%
```

- Analizador sintáctico: bison

```
%token NPAR NIMPAR
%%
axioma: NPAR NIMPAR noter NPAR ;

noter: NPAR NIMPAR
       {printf("%d %d %d %d",
               $-1, $0, $1, $2);}
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2.`

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

1. Bison inicia el análisis pidiendo un tóken a flex.

Metavariables: ejemplo par e impar (paso a paso)

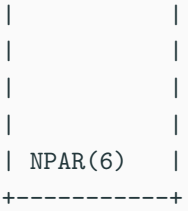
Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

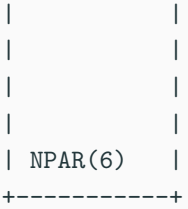
1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

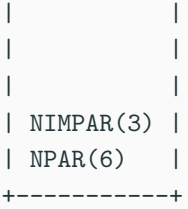
1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

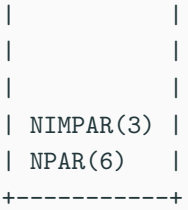
1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

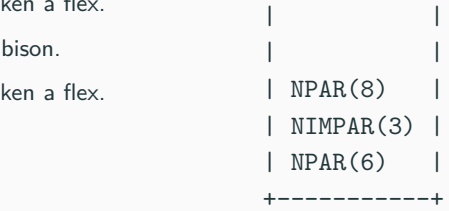
1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el 8, lo coloca en `yylval` y retorna `NPAR` a bison.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

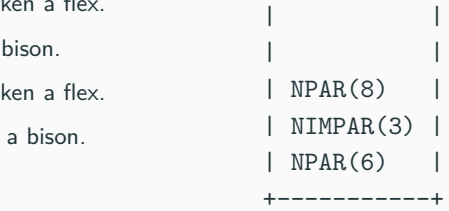
1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el 8, lo coloca en `yylval` y retorna `NPAR` a bison.
7. Bison desplaza a la pila el token y pide un nuevo token a flex.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el 8, lo coloca en `yylval` y retorna `NPAR` a bison.
7. Bison desplaza a la pila el token y pide un nuevo token a flex.
8. Flex lee el 1, lo coloca en `yylval` y retorna `NIMPAR` a bison.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2.`

1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el 8, lo coloca en `yylval` y retorna `NPAR` a bison.
7. Bison desplaza a la pila el token y pide un nuevo token a flex.
8. Flex lee el 1, lo coloca en `yylval` y retorna `NIMPAR` a bison.
9. Bison desplaza a la pila el token y se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`».

```
     |           |
$2   | NIMPAR(1) |
$1   | NPAR(8)   |
$0   | NIMPAR(3) |
$-1  | NPAR(6)   |
     +-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

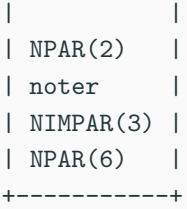
1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el `6`, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el `3`, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el `8`, lo coloca en `yylval` y retorna `NPAR` a bison.
7. Bison desplaza a la pila el token y pide un nuevo token a flex.
8. Flex lee el `1`, lo coloca en `yylval` y retorna `NIMPAR` a bison.
9. Bison desplaza a la pila el token y se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`».
10. Ejecuta la acción semántica, reduce y pide un nuevo token a flex.

```
|           |
|           |
| noter     |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el 8, lo coloca en `yylval` y retorna `NPAR` a bison.
7. Bison desplaza a la pila el token y pide un nuevo token a flex.
8. Flex lee el 1, lo coloca en `yylval` y retorna `NIMPAR` a bison.
9. Bison desplaza a la pila el token y se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`».
10. Ejecuta la acción semántica, reduce y pide un nuevo token a flex.
11. Flex lee el 2, lo coloca en `yylval` y retorna `NPAR` a bison.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

1. Bison inicia el análisis pidiendo un tóken a flex.
2. Flex lee el 6, lo coloca en `yylval` y retorna `NPAR` a bison.
3. Bison desplaza a la pila el token y pide un nuevo token a flex.
4. Flex lee el 3, lo coloca en `yylval` y retorna `NIMPAR` a bison.
5. Bison desplaza a la pila el token y pide un nuevo token a flex.
6. Flex lee el 8, lo coloca en `yylval` y retorna `NPAR` a bison.
7. Bison desplaza a la pila el token y pide un nuevo token a flex.
8. Flex lee el 1, lo coloca en `yylval` y retorna `NIMPAR` a bison.
9. Bison desplaza a la pila el token y se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`».
10. Ejecuta la acción semántica, reduce y pide un nuevo token a flex.
11. Flex lee el 2, lo coloca en `yylval` y retorna `NPAR` a bison.
12. Bison desplaza a la pila el token y se da cuenta de que puede reducir por «`axioma : NPAR NIMPAR noter NPAR`». Reduce y finaliza el análisis.

```
|           |
| NPAR(2)   |
| noter     |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Metavariables: tipos de datos en la pila

Es habitual intercambiar información en la pila de distintos tipos (no solamente enteros como hasta ahora).

Para ello, se puede definir en bison una unión[4] (`%union`) que representará los distintos tipos de datos que se puedan mover a la pila. En este caso, se debería indicar de qué tipo es cada token en su definición en bison.

Además, si se accede a posiciones de la pila de producciones previas, siempre deberá indicarse el tipo de elemento que hay en la pila con: «$<nom_var>num». Donde `nom_var` es el nombre de la variable de la unión y `num` es la posición del elemento en la pila.

[4] Recuerda la diferencia entre «`struct`» y «`union`» en C.

Metavariables: tipos de datos en la pila (ejemplo)

En la siguiente gramática se definen dos variables «p» e «i» (aunque ambas sean del mismo tipo, podrían ser una float y otra char, etc.).

- Analizador léxico: flex

```
 #include "y.tab.h"
%%
[0-9]+[02468]   {yylval.p=atoi(yytext);
                 return NPAR;}
[0-9]+[13579]   {yylval.i=atoi(yytext);
                 return NIMPAR;}




%%
```

- Analizador sintáctico: bison

```
%union {
  int p;
  int i;
}
%token <p>NPAR <i>NIMPAR
%%
axioma: NPAR NIMPAR noter NPAR ;

noter: NPAR NIMPAR
       {printf("%d %d %d %d",
               $<p>-1, $<i>0, $1, $2);};
```

Metavariables: reglas semánticas intermedias

No es necesario que las reglas aparezcan al final de la producción, es posible tener reglas semánticas intermedias, que también pueden almacenar un valor semántico.

En este punto es importante entender que las reglas semánticas (cualquier código entre llaves) tendrá su posición en la pila y esto se debe tener en cuenta:

```
         $1   $2      $3   $4
asigna: ID {$$=$1;} IGUAL NUM {printf("Asigna %d a %s\n", $4, $2);};
```

Metavariables: ejemplo par e impar

A la gramática que se ha visto antes, se le han añadido algunas reglas semánticas nuevas:

- Analizador léxico: flex

```
 #include "y.tab.h"
%%
[0-9]+[02468]   {yylval.p=atoi(yytext);
                 return NPAR;}
[0-9]+[13579]   {yylval.i=atoi(yytext);
                 return NIMPAR;}

%%
```

- Analizador sintáctico: bison

```
%union {
  int p;
  int i;
}
%token <p>NPAR <i>NIMPAR
%%
axioma: NPAR NIMPAR { $<i>$=$1+$2; }
        noter NPAR  {printf("%d\n",
                            $<i>3);};

noter: NPAR NIMPAR
       {printf("%d %d %d %d",
               $<i>-1, $<i>0, $1, $2);};
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2.`

Metavariables: ejemplo par e impar (paso a paso)

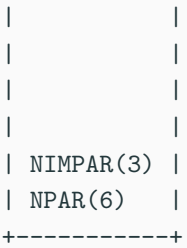
Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison recibe el 6 de flex y lo desplaza a la pila.

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

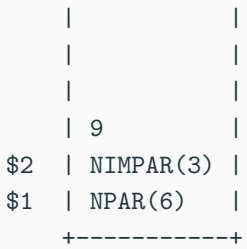
1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.



Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.

```
|           |
|           |
| NPAR(8)   |
| 9         |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.
5. Bison recibe el 1 de flex y lo desplaza a la pila.

```
|           |
| NIMPAR(1) |
| NPAR(8)   |
| 9         |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2.`

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.
5. Bison recibe el 1 de flex y lo desplaza a la pila.
6. Bison se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`»

```
     |           |
$2   | NIMPAR(1) |
$1   | NPAR(8)   |
$0   | 9         |
$-1  | NIMPAR(3) |
     | NPAR(6)   |
     +-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.
5. Bison recibe el 1 de flex y lo desplaza a la pila.
6. Bison se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`»
7. Bison reduce y pide un nuevo token a flex.

```
|           |
|           |
| noter     |
| 9         |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: 6 3 8 1 2.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.
5. Bison recibe el 1 de flex y lo desplaza a la pila.
6. Bison se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`»
7. Bison reduce y pide un nuevo token a flex.
8. Bison recibe el 2 de flex y lo desplaza a la pila

```
|           |
| NPAR(2)   |
| noter     |
| 9         |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.
5. Bison recibe el 1 de flex y lo desplaza a la pila.
6. Bison se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`»
7. Bison reduce y pide un nuevo token a flex.
8. Bison recibe el 2 de flex y lo desplaza a la pila
9. Bison realiza la acción semántica: imprimir la posición 3 de la pila.

```
     |           |
$5   | NPAR(2)   |
$4   | noter     |
$3   | 9         |
$2   | NIMPAR(3) |
$1   | NPAR(6)   |
     +-----------+
```

Metavariables: ejemplo par e impar (paso a paso)

Texto de entrada al analizador sintáctico: `6 3 8 1 2`.

1. Bison recibe el 6 de flex y lo desplaza a la pila.
2. Bison recibe el 3 de flex y lo desplaza a la pila.
3. Bison realiza la acción semántica: deja en el tope de pila el resultado de la suma.
4. Bison recibe el 8 de flex y lo desplaza a la pila.
5. Bison recibe el 1 de flex y lo desplaza a la pila.
6. Bison se da cuenta de que puede reducir por la producción «`noter : NPAR NIMPAR`»
7. Bison reduce y pide un nuevo token a flex.
8. Bison recibe el 2 de flex y lo desplaza a la pila
9. Bison realiza la acción semántica: imprimir la posición 3 de la pila.
10. Bison se da cuenta de que puede reducir por la producción «`axioma : NPAR NIMPAR {...} noter NPAR {...}`», reduce y finaliza.

```
|           |
| NPAR(2)   |
| noter     |
| 9         |
| NIMPAR(3) |
| NPAR(6)   |
+-----------+
```

Tipo de datos de un no terminal

Del mismo modo que a los tókenes se les puede asignar un tipo, a los no terminales también.

Se realiza mediante la directiva %type que funciona de modo análogo a %token.

Ejemplo:

```
%union {
  int p;
  int i;
}
%token <p>NPAR <i>NIMPAR
%type  <p>noter
%%
axioma: NPAR NIMPAR noter NPAR  {printf("%d\n", $3);};

noter: NPAR NIMPAR  {printf("%d %d %d %d", $<p>-1, $<i>0, $1, $2);}
                    {$<p>$ = $1;};
```

4. Gramáticas ambiguas

Resolución de conflictos

En gramáticas ambiguas, pueden aparecer conflictos de tipo desplazamiento/reducción o reducción/reducción.

Bison informa de los conflictos por línea de comandos. Ejemplo:

```
aviso: 1 conflicto reducción/reducción
```

En caso de conflicto:

- desplazamiento/reducción: bison favorece el desplazamiento.
- reducción/reducción: bison reduce por la regla que aparezca primero en el fuente.

Es aconsejable eliminar los conflictos que puedan aparecer para evitar comportamientos no esperados.

Resolución de conflictos: desambiguar gramáticas

Ejemplo de cómo realizar operaciones aritméticas utilizando un único no terminal:

- Analizador léxico: flex

```
%%
[0-9]+   return NUM;

.        return yytext[0];

%%
```

- Analizador sintáctico: bison

```
%token NUM
%%
axioma : exp ;
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | '(' exp ')'
    | NUM ;
```

¿Qué devuelve bison al analizar la gramática?

Resolución de conflictos: desambiguar gramáticas

Ejemplo de cómo realizar operaciones aritméticas utilizando un único no terminal:

- Analizador léxico: flex

```
%%
[0-9]+  return NUM;

.       return yytext[0];

%%
```

- Analizador sintáctico: bison

```
%token NUM
%%
axioma : exp ;
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | '(' exp ')'
    | NUM ;
```

¿Qué devuelve bison al analizar la gramática?

```
aviso: 9 conflictos desplazamiento/reducción
```

Resolución de conflictos: desambiguar gramáticas

Ejemplo de cómo realizar operaciones aritméticas utilizando un único no terminal:

- Analizador léxico: flex

```
%%
[0-9]+  return NUM;

.       return yytext[0];

%%
```

- Analizador sintáctico: bison

```
%token NUM
%%
axioma : exp ;
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | '(' exp ')'
    | NUM ;
```

¿Qué devuelve bison al analizar la gramática?

```
aviso: 9 conflictos desplazamiento/reducción
```

Esto sucede porque bison no conoce la precedencia de cada operador ni si asocian a izquierdas o derechas, ¿cómo se puede solucionar?

Desambiguar gramáticas

Bison dispone de tres directivas que permiten desambiguar gramáticas:

- `%right` Indica que el/los símbolo/s asocia/n a derecha.
- `%left` Indica que el/los símbolo/s asocia/n a izquierda.
- `%nonassoc` Indica que el/los símbolo/s no puede/n aparecer más de una vez en una sentencia. Ejemplos: asignación, mayor o menor que...

Más información sobre precedencia de operadores en el manual de bison.

Desambiguar gramáticas: ejemplo

Ejemplo de cómo realizar operaciones aritméticas con un único no terminal:

- Analizador léxico: flex

```
%%
[0-9]+  return NUM;

.       return yytext[0];

%%
```

- Analizador sintáctico: bison

```
%token NUM
%left '+' '-'
%left '*'
%%
axioma : exp ;
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | '(' exp ')'
    | NUM ;
```

Como se puede ver en el ejemplo, los símbolos definidos primero en el fuente tendrán menor prioridad («*» más prioritario que «+» y «-»).

5. Comandos de interés

Opciones interesantes

Existen diversas opciones interesantes en bison que aún no han sido tratadas:

- Depurar/*debug*.
- Visualizar el diagrama de estados.

Opciones interesantes: depurado

Bison permite realizar depuración/*debug* sobre el código generado. Para ello se debe:

1. Agregar en la zona de definiciones de C la siguiente línea: `int yydebug = 1;`
2. Compilar con la opción `-DYYDEBUG`. Es decir, ejecutar `gcc` indicando al final dicha opción (tras las banderas `-lfl`).

Ejemplo:

```
gcc lex.yy.c y.tab.c -lfl -DYYDEBUG
```

Opciones interesantes: diagrama de estados

Bison permite visualizar el diagrama de estados, que genera internamente para el análisis (tablas LALR), de dos formas:

- Modo texto: `bison -v analizador_sintactico.y`.
- Modo gráfico: permite exportar a pdf, html, etc.

Ejemplos:

- Exportar a pdf:

```
bison -yg eje4.y -b eje4
dot -Tpdf eje4.dot -o eje4.dot.pdf
```

- Exportar a html:

```
bison -yx eje4.y
export datadir=`bison --print-datadir`
xsltproc $datadir/xslt/xml2xhtml.xsl y.xml > eje4.html
```

6. Manejo de errores

Manejo de errores

En JavaCC, cuando el analizador localizaba un token que no se encontraba en el lugar que la gramática esperaba, devolvía un error indicando dónde se había producido el error y por qué token en concreto.

Hasta ahora, hemos visto que bison también falla cuando detecta un token inesperado.

No obstante, este comportamiento solo permite detectar un único fallo en la entrada: el que aparezca en primer lugar.

Para ello bison dispone de las «producciones de error».

Producciones de error

Como se indicó al inicio, la palabra `error` esta reservada en bison (se puede ver como un no terminal que siempre está definido). Es la que utiliza para definir las producciones de error.

Una producción de error es aquella que tiene en el consecuente la palabra `error`. Ejemplo:

```
stmt: loop error ';' ;
```

Una producción de error marca un contexto en el que los tókenes erróneos se pueden eliminar hasta que se detectan tókenes de sincronización apropiados, de donde el análisis sintáctico se puede reiniciar.

Las producciones de error permiten al programador marcar manualmente aquellos no terminales cuyas entradas *lr_a* vayan a utilizarse para recuperación de errores.

Referencias i

Manual de bison, 2020.
`https://www.gnu.org/software/bison/manual/html_node/index.html`, **[Online; accedido 14 de julio de 2020].**

C. García-Osorio.
Apuntes de las prácticas de yacc/bison.