Yacc

- Es un utilitario que permite generar parsers.
- Recibe un archivo en el que se define una gramática, similar a BNF y las acciones semánticas a realizar (en código C)
- Presupone un escáner al que puede llamar invocando la función yylex. Puede trabajar con lex pero NO es un requisito.
- Yacc está discontinuado, la herramienta que se usa hoy dia se llama bison.
- Bison es compatible hacia atrás con yacc

Bison

- La especificación de bison tiene a grandes rasgos el siguiente formato:

```
%{
Prólogo
%}
Declaraciones Bison
%%
Reglas gramaticales
%%
Epílogo
```

- En el prólogo definimos variables y funciones que después usaremos en las acciones de las reglas gramaticales. También incluiremos los encabezados .h que sen necesarios.

Bison

- Las declaraciones bison permiten definir elementos de la gramática como los símbolos terminales y no terminales, la precedencia de operadores y el tipo de valor semántico de cada símbolo.
- Las reglas gramaticales es donde definimos la gramática con anotaciones que queremos analizar, con la notación propia de bison
- El epílogo puede contener código de funciones auxiliares o un main si es un programa simple.

Prólogo

- Igual que en flex podemos incluir código encerrado entre `%{` y `%}`. Se puede tener varios bloques de código intercalados con declaraciones bison.
- Algunas declaraciones pueden necesitar cierto código definido antes y puede haber código que necesite que una determinada declaración bison se haya hecho antes. Esto es frecuente en el caso de la declaración del tipo de dato de los registros semánticos.

Prólogo

- En las versiones recientes se utiliza `%code{` y `}` para encerrar código, en lugar de `%{` y `%}` (aunque siguen siendo válida su utilización) y agregaron `%code` *`calificador`* `{ ... }` donde *`calificador`* puede ser:
  - `top`: para incluir algo bien al principio del fuente
  - `requires`: aquí debemos poner el código que puede ser necesario para las directivas de bison, por ejemplo tipo de datos para los registros semánticos
  - `provides`: aquí van declaraciones que formarán parte del encabezado .h que genere bison para otros módulos.

Declaraciones bison

- Algunas declaraciones comunes:
  - `%defines "archivo.h"` nombre del .h a generar
  - `%output "archivo.c"` nombre del fuente a generar
  - `%start noterminal` indica que noterminal es el axioma de la gramática
- Tokens
  - Se declaran con `%token`, por ejemplo
    - %token INICIO FIN ID CTE
  - Bison genera un enum con los valores de estos token, al incluir el header de bison en flex tenemos acceso a esta constantes para poder devolverlas según regla que aplique

Registros semánticos

- En principio, si no declaramos nada es int
- `%define api.value.type {un-tipo}` indica que el registro semántico es de tipo `un-tipo`, por ejemplo `double` o `char *`
- Si se necesitan diferentes tipos de datos para distintos tokens se usa la declaración `%union`, por ejemplo

```
%union {
  int ival;
  char *sval;
}
```

Luego se indica que token lleva cada tipo de valor, haciendo referencia al nombre del campo entre < y >, por ejemplo

```
%token <ival> CTE
%token <sval> ID
```

Registros semánticos

- El tipo de datos del registro semántico es YYSTYPE, por eso si queremos por ejemplo tener un struct basta con declararlo. Necesitamos que flex conozca esta declaración, así que debemos asegurarnos esté en el header producido por bison, por ejemplo:

```
%code provides {
   struct YYSTYPE {
      char  *lexem;
      int   value;
   };
}

%define api.value.type {struct
YYSTYPE}
```

Registros semánticos

- En flex se puede colocar datos en el registro semántico mediante la variable `yylval`
- Con un único tipo de valor, por ejemplo un int, simplemente hacemos `yylval = 3;`
- Si tengo union o struct deberé acceder al campo que corresponda. Para el ejemplo anterior en flex tendríamos algo como:

```
yylval.lexem = strdup(yytext);
sscanf(yytext,"%d",&yylval.value);
return CTE;
```

Reglas gramaticales

- Bison usa una BNF donde `:` es el metasímbolo de producción, `|` permite poner más de una producción para el mismo no terminal a izquierda y se usa `;` para indicar el final de las producciones para un determinado no terminal, por ejemplo:

```
programa : INICIO lsentencias FIN ;
lsentencias : sentencia
            | lsentencias
              sentencia
              ;
```

- Por convención, ya que los terminales los escribimos en mayúsculas, los no terminales van en minúsculas

Reglas gramaticales

- Notar el uso de recursión en el ejemplo anterior de `lsentencia`. Bison puede manejar tanto recursión a izquierda como a derecha, sin embargo es recomendable tratar de evitar la recursión a derecha dado que es lenta y consume más memoria.
- En caso que una producción sea ε se deja vacía, o se pone un comentario (/* epsilón */) o bien se usa la declaración %empty

```
noterm1 : | noterm1 UNTOKEN ;
noterm2 : %empty | ALGO ;
```

Reglas gramaticales y tokens

- No es necesario declarar todos los tokens, en particular aquellos que se componen de un único carácter puedo usarlos entre comillas simples:
  - `exp : exp '+' exp ;`
- En flex devolvemos como token el carácter que usamos, por ejemplo:
  - `{ return '+' ;}`
- Al declarar un token podemos asociarle un literal cadena, y luego usar indistintamente uno u otro en las reglas gramaticales:
  - `%token ASIGNACION ":="`
  - `sentencia  : identif ":=" exp ';'`

Precedencia y asociatividad

- Bison permite escribir una BNF "achatada" que en principio sería ambigua, pero resuelve el problema indicando precedencia y asociatividad por separado.
- Puedo indicar la asocatividad y precedencia de un operador declarando su token, pero cambiando `%token` por `%left`, `%right` o `%nonassoc`, donde este último indica que es un error sintáctico una construcción del tipo '`x op y op z`'
- Los operadores que declaro primero tienen menor precedencia que los últimos
- Los que están a mismo nivel comparten la precedencia

Precedencia y asociatividad

- Para un subconjunto de lenguaje C podría tener las siguientes declaraciones:

```
%right '='
%left '<' '>'
%left '+' '-'
```

- Hay casos donde quiero declarar precedencia pero sin asociatividad, para eso usamos `%precedence`
- También es útil para cambiar la precedencia de un operador cuando se usa en otro contexto, por ejemplo unario en lugar de binario.

Precedencia y asociatividad

- Para distinguir el '-' binario (resta) de unario (cambio de signo) defino primero las precedencias

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

%precedence NEG
```

- Notar que NEG es un token que no uso (flex nunca devuelve NEG) solo sirve para indicar la precedencia, entonces luego en las reglas gramaticales uso ese nombre para indicar "este '-' lleva la precedencia de NEG y no la propia"

```
exp: /* otras reglas */ ...
        | exp '-' exp
        | '-' exp  %prec NEG
```

- El modificador `%prec NEG` (que se coloca al final de la regla) hace que ese `'-'` tenga más precedencia que por ejemplo un `'*'` ya que `NEG` fue declarado posteriormente

Acciones

- En cada regla gramatical vamos a insertar acciones. La idea es manejar los registros semánticos para realizar controles semánticos o para generar el código intermedio.
- Vimos como asociar un determinado tipo de valor semántico, por ejemplo cuando tengo una unión con los posibles tipos, a un token. Para asociar a un no terminal usamos `%type`, por ejemplo:
  - `%type<sval> exp`
- El valor semántico del lado izquierdo de la producción se referencia como $$ en tanto que los del lado derecho según su aparición como $1, $2, etc.
- En caso de usar un struct puedo indicar que campo uso:
  - $1.sval $$.ival

Acciones

- En caso de usar un `%union` puedo indicar que campo uso:
  - `$<sval>1  $<ival>$`
- Las acciones suelen colocarse al final de la regla gramatical, pero también pueden colocarse en medio de la misma. En ese caso no puede referenciarse el valor semántico de la parte izquierda de la producción.
- Si los nombres no se repiten puedo usarlos en lugar el número, así en lugar de:
  - `resultado : ID ':' NUM {$$ = $3;}`
- Donde estoy asignando al registro semántico de resultado el valor del registro semántico del tercer elemento de la producción, puede escribirlo como:
  - `resultado : ID ':' NUM {$resultado = $NUM;}`

Acciones

- En caso que los nombres se repitan puedo desambiguar con un alias en corchetes
  - `exp[resul] : exp[izq] '*' exp[der]`
    `{$resul = $izq * $der;}`
- Al margen de estos ejemplos elementales puedo llamar funciones en tanto las haya declarado previamente y usar los registros semánticos como parámetros o para recibir resultados.
- El uso del | para separar reglas es un acortador de notación, se consideran reglas con la misma parte izquierda, pero separadas.
- Si en una regla no ponemos ninguna acción, bison llama la acción por defecto que es `$$ = $1;`

Invocación

- La rutina para llamar al parser generado por bison tiene la declaración: `int yyparse (void)` y devuelve:
  - 0 si tuvo éxito
  - 1 si hay errores sintácticos
  - 2 si no alcanzó la memoria ram para ejecutar
- No hace falta agregar una regla objetivo explícitamente ya que bison la provee, si nuestro axioma es programa, bison agrega la regla
  - `$accept: programa $end`
- donde `$accept` y `$end` son no terminales predefinidos por bison (que no puede ser usados en la gramática). $end se activa cuando flex envía EOF (lo hace automáticamente, no es necesario una regla del tipo `<<EOF>> {return EOF;}`

Manejo de errores

- Bison espera que se declare y defina la función yyerror. Para los casos simples que trataremos aclanza con declararla como:
  - `void yyerror (char const *msg);`
- Bison llamará esta función con el parámetro "syntax error" al encontrar un error sintáctico. Si queremos mayor información en el mensaje lo indicamos de dos modos posibles
  - En el prólogo podemos agregar: `#define YYERROR_VERBOSE`
  - O usar la declaración: `%define parse.error verbose`

Manejo de errores

- En la variable global `yynerrs` bison lleva la cantidad de errores sintácticos encontrados. Los errores léxicos deberemos contarlos por nuestra cuenta. Desde flex puedo informar los errores llamando a `yyerror` si incluí el header producido por bison.
- Si en bison (o el fuente donde defina la función yyerror) incluí el header generado por flex puedo usar la variable `yylineno` en la descripción del error.
- Al encontrar un error sintáctico `yyparse` llama a `yyerror` luego de lo cual regresa con resultado 1 para indicar que falló, es decir que no continúa analizando el fuente

Recuperación de errores

- Para evitar que bison detenga el análisis al primer error se debe incluir en la gramática el no terminal, predefinido en bison, llamado `error` el que puede ser usando en las reglas, por ejemplo:

```
line: '\n'
        | exp '\n'
        | error '\n'
```

- Si un error ocurre durante el análisis de `exp` bison descartará de su pila de análisis los símbolos de exp de modo de llegar a emparejar la última regla (`error '\n'`) y si es necesario descartará símbolos posteriores de modo de llegar al `'\n'`.
- En definitiva la última regla indica que si detecta un error trate de resincronizar con el sigiuente `'\n'` y luego continuar el análisis.

Recuperación de errores

- El uso del no terminal `error` no evita la llamada a `yyerror` pero desactiva subsiguientes llamadas, que normalmente, son errores por culpa del primer error. Las llamadas a `yyerror` se reanudan luego de 3 símbolos leídos con éxito (típicamente al resincronizar, en este caso con el `'\n'`).
- Si se desea no suspender las llamadas a `yyerror` se puede usar en la acción de la regla de error la macro `yyerrok;` (no aconsejado, normalmente confunde más de lo que ayuda)

Recuperación de errores

- El uso del terminal `error` tiene la ventaja de recuperarnos del error, y la desventaja que `yyparse` muy probablemente termine devolviendo 0 (éxito).
- Un modo de solucionarlo es con una acción al final del último símbolo que controle la cantidad de errores (`yynerrs` serguirá incrementándose) y llamar a `YYABORT` para retornar de yyparse con resultado 1 o a `YYACCEPT` para retornar con resultado 0
- YYERROR es una macro que puedo invocar en la acción de una regla gramatical que provoca que se inicie una recuperación de error “como si” hubiese habido un error sintáctico, pero no incrementa `yynerrs`.
- Es útil para cuando detectamos un error semántico, por ejemplo el uso de una variable no declarada en una sentencia, para evitar que siga analizando dicha sentencia

Uso de bison

- para invocar bison basta con
  - `bison definicion.y`
- O bien
  - `bison -d definicion.y`
- Suponiendo que no usamos las directivas `%output` y `%defines` se generará el fuente `definicion.tab.c` y solo usando -d el encabezado `definicion.tab.h`
- Suponiendo que en una misma carpeta tengo mis definiciones de flex y bison junto a otros fuentes, puedo armar el compilador completo con los siguientes comandos

```
flex scanner.l
bison parser.y
gcc -Wall *.c -o compilador -lfl
```

Licencia