Lex

- Es un utilitario que permite generar escáneres y programas de transformación de texto.
- Toma una especificación como entrada, que ya describiremos, y genera como salida un fuente en lenguaje C con el código del escáner
- Lex está discontinuado, la herramienta que se usa hoy dia se llama flex.
- Flex tiene algunas incompatibilidades menores con lex

Flex

- La especificación de flex tiene el siguiente formato:

```
definiciones
%%
reglas
%%
código del usuario
```

- La sección de definiciones sirve para dar nombres a ciertas ER y así simplificar y hacer más claras otras ER.
- La sección de reglas se delimita con %% y es donde indicamos mediante ER los tokens a reconocer y mediante código C las acciones a realizar cada vez que se reconoce un token.
- La sección de código de usuario es optativa. En caso de no estar el segundo delimitador %% no es necesario. Sirve para agregar código que flex copiará tal cual al final del fuente que genere.
- Puede haber una función main que típicamente haga al menos un llamado a yylex() o bien puede haber funciones auxiliares a ser invocadas por la acciones de las reglas

Definiciones

- La Sintaxis es:
  - nombre definición
  - Nombre debe estar al comienzo de la línea y comenzar con una letra o un guión bajo, continuando luego con más letras, guiones bajos, guiones del medio (menos) o dígitos.
  - Definición se extiende desde el primer carácter no blanco luego del nombre hasta el fin de la línea, siendo una ER que eventualmente puede usar nombres definidos previamente.
- Ejemplo:
  - `digito [0-9]`
  - `exponente [eE][+-]?{digito}+`
- Uso en las reglas
  - `{digito}+   {nro++; //esta es la accion}`

Definiciones

- Para usar un nombre lo coloco entre llaves. Se reemplaza {nombre} con (definición) siendo los paréntesis para evitar problemas de precedencia
  - Si la definición del nombre incluye `^` o `$` para indicar principio o fin de línea, la expansión es sin paréntesis para preservar el significado
    - En esto se diferencia de lex (que nunca pone paréntesis)
  - No puede usarse en definición: `<ci>` (condición inicial), `<<EOF>>`, `/` (operador de contexto)

Otros en Definiciones

- Se copia textualmente al fuente resultante:
  - Comentarios C no indentados
  - Texto indentado (código C)
  - Bloques de texto encerrados entre `%{` y `%}` estos delimitadores deben estar NO indentados
  - Un bloque top, que se delimita con `%top{` y `}` , es similar al anterior pero se asegura que irá al tope del fuente generado. Si hay varios se respeta el orden entre ellos
- Opciones, pueden ir en la zona de definiciones o en la línea de comando que invoca a flex
  - `%option header-file="archivo.h"` nombre del .h a generar
  - `%option outfile="archivo.c"` nombre del fuente a generar
  - `%option yylineno` indica que mantega el número de línea

Reglas

- La Sintaxis es:
  - `patrón acción`
    - **Patrón** es una ER que no debe estar indentada y se separa de la acción por el primer blanco (espacio o tabulador no encerrado entre comillas o con barra invertida delante).
    - **Acción** es código C que debe comenzar en la misma línea. Si es una sentencia compuesta puede extenderse a las líneas siguientes.
- Cada vez que ejecuta el escáner trata de emparejar el texto entrante con todos los patrones. Si puede lograr emparejar con más de una patrón el orden de precedencia es:
  - El patrón **más largo**, es decir el que tiene más caracteres
    - Ejemplo si tengo en patrón **`for`** (que solo encuentra la cadena for) y el patrón **`[a-z]+`** y en la entrada el escáner lee **forever** aplica la acción correspondiente al segundo patrón.
  - Si dos patrones emparejan con la misma cantidad de caracteres, aplica el que se haya **definido primero**.

Reglas

- Si después del inicio de la sección de reglas, marcada por %% y antes de la primer regla se coloca código indentado o entre %{ y %} (estos delimitadores NO identados) dicho código se coloca al inicio de la rutina del escáner.
- Así es posible declarar variables locales a la rutina del escáner y/o código que se ejecutará siempre al inicio de la rutina dicha rutina
- Si agrego código después de la primer regla no está definido donde será colocado y suele terminar dando errores de compilación (evitar, está no más porque posix lo pide)

Patrones

- Las ER que usa flex son muy similares a las del programa egrep. Coincidencias:

Operador	Comentario
.	Cualquier carácter menos \n y EOF
\t \n \0123 \x2a … etc	Secuencias de escape propias de C
\+ * \. \(… etc	Secuencias de escape para los operadores de ER
|	Pipe para unión
* +	Clausura de Kleene, clausura positiva
?	0 o 1 repetición
[ ] [^] [-]	1 carácter del conjunto
{n} {n,m} {n,}	Potencia
( )	Paréntesis para modificar precedencias
^ $	Al inicio o al fin de la línea
[:digit:] [:^digit:] … etc	Como en isdigit de ctype.h y su complemento. Vale para otros isXXX (debe volver a encerrarse entre [])

Patrones

- Algunas particularidades de flex:

Operador	Comentario
<<EOF>>	Empareja con EOF
r/s	ER r siempre y cuando esté seguida inmediatamente por la ER s. Se empareja y devuelve r pero se toma el largo de rs para seleccionar que patrón seleccionar si más de uno empareja.
{-}	Para restar conjuntos: [a-z]{-}[xf] encuentra cualquier minúscula salvo x o f
{+}	Similar al anterior pero hace unión de conjuntos
<ci>r	Encuentra r solo si se está en Condición Inicial ci
<ci1,ci2,ci3>r	r si la condición inicial es ci1 o ci2 o ci3
<*>r	r en cualquier condición inicial
"." "a+" … etc	Lo que encierro entre comillas dobles es literal, por ejemplo una a seguida de un +, y no la clasura positiva de a

Acciones

- Indican que hacer cuando se encuentra el patrón asociado a las mismas
  - Ejemplo: `\n caracteres++; lineas++;`
  - Al encontrar un \n incrementa dos contadores
- Puede ser la sentencia nula o un comentario para indicar que no se hace nada, simplemente se descarta el patrón encontrado
- Si no se puede emparejar ningún patrón se aplica la acción por defecto, esto es tomar el primer carácter de entrada y copiarlo tal cual en la salida
- Para copiar un patrón emparejado, tal cual en la salida se usa la directiva ECHO (seguido de un ;)
- Si se usa un | (pipe) como acción lo que se está indicando es que debe realizar la misma acción que el patrón siguiente.

Código de Usuario

- Es código que se agregará al final de fuente generado por flex
- Hay dos casos típicos
  - Agregar una rutina main que llama al escáner. Es el caso cuando se quiere hacer un programa que procese un texto, para modificarlo y/o hacer estadísticas sobre el mismo.
  - Agregar rutinas auxiliares que puedan ser invocadas desde las acciones, por supuesto habrá que declararlas antes, por ejemplo, agregando código en la sección de definiciones

Condiciones Iniciales

- Permiten definir “sub-escáneres”, es decir cambiar el modo de escaneo. Es útil cuando las reglas cambian, por ejemplo en comentarios o en literales cadena.
- Se los declara en la sección de definiciones con %s si es una condición inclusiva o %x si es una condición exclusiva
  - Ejemplos:
    - `%s CODIGO DATOS`
    - `%x CADENA`
- Hay una condición predefinida llamada INITIAL cuyo valor es cero, si una regla no tiene condición inicial, tiene implícitamente la condición INITIAL

Condiciones Iniciales

- Para activar una condición inicial se usa en la acción la directiva BEGIN(condición)
- La condición queda activa hasta que se activa otra en su reemplazo, por ejemplo BEGIN(INITIAL)
- Cuando una condición exclusiva está activa solo las acciones con esa condición se toman en cuenta para emparejar patrones.
- Cuando una condición inclusiva está activa se consideran la acciones con esa condición y las acciones sin condición inicial.

Variables y funciones disponibles

- **yylex()** invoca al escáner
- **yytext** tiene el lexema encontrado, en modo flex es un char * (solo accesible desde flex, se debe copiar en un registro semántico para ser usado en bison)
- **yyleng** largo de yytext
- **yyin** es el FILE * de donde el escáner lee la entrada, por defecto stdin
- **yyout** es el FILE * a donde el escáner dirige su salida, por defecto stdout

Uso Elemental

- Si mi archivo de configuración se llama prueba.l armo el fuente C con
  - `flex prueba.l`
- Compilo teniendo en cuenta de agregar la biblioteca de lex
  - `gcc -o prueba prueba.c -lfl`
- Ejecuto usando el achivo prueba.txt como entrada
  - `./prueba <prueba.txt`

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.
- 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 gramáticales 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.
- En las versiones recientes se utiliza `%code{` y `}` para encerrar código, en lugar de `%{` y `%}` (aunque siguen siendo válida su utilización) y %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}
```

- 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
- Note el uso de recursión en el segundo ejemplo. 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.
- 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 cambiando `%token` por `%left`, `%right` o `%nonassoc`, donde el 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
- 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.

Ejemplos de precedencia y asociatividad

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

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

- 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`
- 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.

Acciones

- Si los nombres no se repiten puedo usarlos en lugar el número, así en lugar de:
  - `resultado : ID ':' NUM {$$ = $3;}`
- Donde estoy asignando al registro semántico de resultado el valor del registro semántico del tercer elemento de la producción, puede escribirlo como:
  - `resultado : ID ':' NUM {$resultado = $NUM;}`
- En caso que los nombres se repitan puedo desambiguar con un alias en corchetes
  - `exp[resul] : exp[izq] '*' exp[der] {$resul = $izq * $der;}`
- Al margen de estos ejemplos elementales puedo llamar funciones en tanto las haya declarado previamente y usar los registros semánticos como parámetros o para recibir resultados.
- El uso del | para separar reglas es un acortador de notación, se consideran reglas con la misma parte izquierda, pero separadas.
- Si en una regla no ponemos ninguna acción, bison llama la acción por defecto que es `$$ = $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`
- En la variable global `yynerrs` se lleva la cantidad de errores sintácticos encontrados. Los errores léxicos deberemos contarlos por nuestra cuenta. Desde flex puedo llamar a `yyerror` si incluí el header producido por bison.
- Si en bison incluí el header generado por flex puedo usar la variable `yylineno` en la descripción del error.

Recuperación de errores

- Al encontrar un error sintáctico `yyparse` llama a `yyerror` luego de lo cual llama a regresa con resultado 1 para indicar que falló, es decir que no continúa analizando el fuente.
- Para evitar esto bison declara un no terminal llamado `error` el que puede ser usando en las reglas, por ejemplo:

```
stmts: stmts '\n'
     | stmts exp '\n'
     | stmts 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 (`stmts 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;`
- 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 a pesar del no terminal `error`) y llamar a `YYABORT` para retornar de yyparse con resultado 1 o a `YYACCEPT` para retornar con resultado 0

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 *.c -o compilador -lfl
```

Licencia