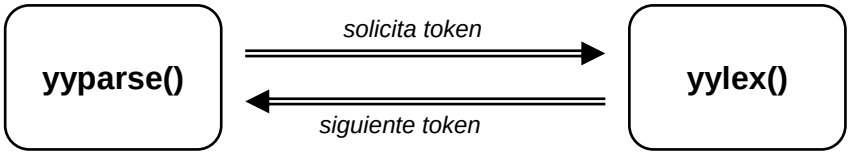
I. Introducción

La sigla *Yacc* proviene de "Yet Another Compilers Compiler".
Está herramienta está escrita en lenguaje C y genera un compilador para un lenguaje determinado. Utiliza el método de parsing ascendente LALR.
Incluye

a) reglas sintácticas que describen la estructura del lenguaje a analizar
b) código a ser invocado cuando esas reglas son reconocidas
c) código auxiliar para el soporte del código de entrada

Yacc lleva adelante el proceso de análisis sintáctico a través de la función especia,l propia de yacc, **yyparse()** que invocará una función especial para controlar el proceso de entrada. Esta función deberá ser llamada **yylex()** y podrá interactuar con **yyparse()** cuando esta última necesite analizar los tokens del programa fuente original de acuerdo al flujo de entrada.



Cuando una regla se reconoce se dispara una acción, que podrá ser implementada a través de código.

II. Estructura – Secciones – Especificaciones básicas

Un programa Yacc se organiza en 3 grandes secciones

1) Declaraciones
2) Tokens, start
3) Definición de reglas
4) Código

1. Sección Declaraciones:

Esta sección deberá incluir todas las declaraciones (inlcudes, defines, variables globales, que se utilizarán en el código y deberá enmarcarse por lols signos **%{ %}**

Ejemplo:

```
%{
      #include <stdio.h>
      #define …
      int matrizEstados [34] [22]
      int ( *proceso[34] [22] ) (void)
      int q
      typedef pila_ass* pilaAss;
      .
 %}
```

2. Sección Tokens:

Esta sección deberá incluir todas las declaracionesde tokens, asociatividades y definición de start symbol.

No es obligatorio declarar el start symbol explícitamente dentro de la sección de declaraciones. Si no se declara se tomará el no terminal que se encuentre en la primera regla escrita en la sección de reglas

Ejemplo

```
%token ID
%token CTE
%left
%start programa
```

3. Sección Definición de Reglas:

Esta sección deberá incluir todas las definiciones de las reglas gramaticales necesarias para llevar adelante el análisis sintáctico y deberá enmarcarse por los signos %**%.....%%.**
Los nombres de los elementos no terminales pueden tener una longitud arbitraria y pueden estar conformados por letras, puntos, "underscore" y dígitos, aunque no pueden empezar con dígitos. Mayúsculas y minúsculas son indistintas.
Los elementos terminales deberán corresponderse con los objetos definidos en la sección declaración de tokens *(Ver Sección Definición Tokens)*
Si hay varias reglas gramaticales con el mismo no terminal en su lado izquierdo, la barra vertical "|" puede ser utilizada para representar varias opciones sobre el mismo no terminal.
Todas las reglas deben finalizar con un **;**

Ejemplo:
Las siguientes reglas

expresion: expresion MAS termino;
expresion: expresion MENOS termino
expresion: PARENTA expresión PARENTC;

Pueden ser escritas como sigue:

expresion: expresion MAS termino
| expresion MENOS termino
| PARENTA expresión PARENTC;

(%token PARENTA, PARENTC, MAS , MENOS)

4. Sección Código

Esta última sección deberá incluir todo el código necesario para la ejecución del programa

Ejemplo

```
int yylex(void)
{
.
.
.
      yyval = posicion en tabla de simbolos(token)
      return = token

{
```

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

int main (int argc, char *argv[])
{

    while (feof(archivo)== 0)
    {
      yyparse();
    }
}
```

III. Acciones

Las acciones son ejecutadas cada vez que una regla es reconocida con el proceso de entrada. Esas acciones pueden retornar valores y pueden obtener valores retornados por acciones previas.

Una acción es una sentencia arbitraria escrita en el lenguaje que compila **yacc**. Deben aparecer encerradas entre llaves **"{" "}"** y finalizada con un **;**

Ejemplo :

```
expresion: expresion OPMAS termino { generarPolaca();}
           expresion OPMENOS termino {generarPolaca();}|
           termino {printf("termino \n");};
```

En las acciones es posible asignar o retornar un valor. Existen variables propias o pseudo variables de yacc a las que se les puede asignar o retornar un valor.
Ejemplo :

Para obtener los valores retornados por las acciones previas y el analizador léxico, la acción puede usar las pseudo variables $1, $2.... que se refieren a los valores retornados por los componentes del lado derecho de una regla, leídos de izquierda a derecha y la variable $$ que se refiere al valor retornado por el componente del lado izquierdo.

Ejemplo
Una acción que no hace nada pero retorna el valor 1 se escribe como:

```
     { $$ = 1; }
```

Sea la siguiente regla

```
     A : B C D ;
```

$1 tiene el valor retornado por B, $2 tiene el valor retornado por C y $3 tiene el valor retornado por D.

Como ejemplo más concreto podemos considerar la siguiente regla:

```
expr : PARENTA expr PARENTC;
```

El valor retornado por esta regla es, de manera usual, el valor de la expr entre paréntesis. Esto se indica como:

```
expr : PARENTA expr PARENTC { $$ = $2; }
```

Por defecto, el valor de una regla es el valor del primer elemento y no necesita una acción explícita.

Por ejemplo:

A : B ;

Frecuentemente no necesita tener una acción explícita.

IV. Ambigüedades y conflictos

Un conjunto de reglas gramaticales (sintácticas) son ambiguas si hay alguna cadena de entrada que puede ser estructurada de 2 o más maneras distintas. Por ejemplo en la gramática:

expr : expr MENOS expr

Si la entrada es la siguiente:

expr - expr - expr

la regla permite estructurar la entrada de este modo
(expr - expr) - expr

o de este otro modo
expr - (expr - expr)

La primera es llamada asociación a izquierda y la otra a derecha.
Yacc detecta esta ambigüedad cuando intenta efectuar el parsing. Esto es un problema con el cual debe enfrentarse al encontrar entradas de este tipo y deberá ser solucionado.

V. Precedencia

La precedencia y las asociaciones son asignadas a los *tokens* en la sección de declaraciones. La manera en la que esto se realiza es utilizando las palabras: %left, %right, or %nonassoc al comienzo de la línea, seguida por una lista de *tokens*.
Todos los *tokens* en la misma línea tendrán el mismo nivel de precedencia y asociatividad. Ejemplos:

```
%left MAS MENOS
%left MUL DIVISION
```

Este ejemplo describe la precedencia y la asociatividad de los 4 operadores matemáticos. El más y el menos son asociativos a izquierda y tienen menos precedencia que la división (/) y la multiplicación (*) que también son asociativos a izquierda.
La palabra %right es utilizada para describir operadores con asociación a derecha y la palabra %nonassoc se utiliza para operadores que no puedan ser asociativos por si solos, como es el caso del operador .LT. en Fortran

A .LT. B .LT. C

Es ilegal en Fortran, y como operador puede ser descrito con la palabra %nonassoc en Yacc.

Otros ejemplos:

```
%right IGUAL
%left MAS MENOS
%left MUL DIVISION

%%

expr  :  expr IGUAL expr
      |  expr MAS expr
      |  expr MENOS expr
      |  expr MUL expr
      |  expr DIVISION expr
      |  NAME;
```

Esta regla permite manejar la siguiente sentencia

```
a = b = c*d - e - f*g
```

de la siguiente manera:

```
a = ( b = ( ( (c*d) - e)  -  (f*g) )  )
```

VI. Compilación

Todas las especificaciones vistas en los puntos anteriores deberán incluirse en un archivo de extensión ".y" , por ejemplo "miCompilador.y". Este archivo se compilará con el comando

```
pcyacc miCompilador.y
```

Si el resultado de la compilación es satisfactorio deberán generarse los archivos:

- miCompilador.c (archivo que contiene el parsing del programa original del miCompilador)
- miCompilador.h que contiene las redefiniciones (#define) de los tokens del compilador

Si el resultado de la compilación contiene errores, se informarán los mismos como cantidad de S/R y de R/R (shift reduce o reduce reduce). En este caso deberá compilarse con la opción

```
pcyacc –v miCompilador.y
```

y se obtendrá el archivo **miCompilador.lrt** donde se podrá obtener una lista completa de los errores de ambigüedad correspondientes.