UTN - FRBA
Sintaxis y Semántica de los Lenguajes
Trabajo Práctico Integrador (TP 4) - Analizador léxico, sintáctico y semántico de ANSI C

Enunciado

Implementar en lenguaje C un programa que a partir de la entrada de un archivo fuente *.i* (archivo *.c* preprocesado)[1] **realice el análisis léxico, sintáctico y semántico de ANSI C (*C89/C90*).** El mismo deberá producir como salida en pantalla (salida estándar, *stdout*) un *reporte*.

Para desarrollar el programa se deberá utilizar *Bison* junto con *Flex* para la generación del código C que implemente un analizador léxico (*scanner*), sintáctico (*parser*) y semántico (*rutinas semánticas y TS*) que cumpla con lo requerido.

Estructuras sintácticas mínimas que se espera que se acepten (*las mismas del TP3*):

- **Expresiones:** primarias, post-fijas (*invocaciones de funciones, operadores ++ y -- postfijos*), unarias (*operadores ++ y -- prefijos*), multiplicativas (*operadores * y /*), aditivas (*operadores + y -*), relacionales (*operadores < , > , <= y >=*), de igualdad (*operadores == y !=*), lógicas AND (*operador &&*), lógicas OR (*operador ||*) y de asignación (*operadores = , += , -= , *= y /=*).
- **Declaraciones:** variables (*simples*) y prototipos de funciones (*simples*)
- **Sentencias:** de expresión, compuestas, de selección *(if, else switch)*, de iteración *(while, do/while, for)*, etiquetadas *(case y default)* y de salto (*continue, break y return*).
- **Definiciones externas** (*globales*)**:** definiciones de funciones (*simples*) y declaraciones (*globales*)

Validaciones semánticas mínimas esperadas:

- **Control de tipos de datos** simples de la operación binaria * (multiplicación) utilizando TS cuando alguno de los operandos sea un identificador.
  - La validación de los demás operadores binarios y unarios es opcional.
- **Control de declaración de símbolos** (variables y funciones) utilizando TS
  - Las funciones pueden ser doblemente declaradas (uno o más prototipos de función) mientras no haya conflicto de tipos (que sus tipos de dato no coincidan), ya que implícitamente utiliza el especificador de tipo *extern*.
  - Los identificadores en los parámetros en prototipos de función son opcionales y no se requiere que también coincidan con los otros prototipos de función equivalentes o de una definición de función (en la que sí es obligatorio indicar un identificador).
  - Las funciones y variables no pueden ser doblemente definidas.
  - Colisión de símbolos: identificadores declarados sobre el mismo namespace (ej. identificadores, etiquetas) y con el mismo alcance/scope (ej. globales).
- **Validación de invocación a funciones**: que el identificador corresponde a un símbolo declarado de función y que la cantidad y tipos de parámetros es válida utilizando TS.
- **Validación de asignación** simple (valor-L modificable y tipos de datos) utilizando TS.
- **Validación de sentencias de salto *return*** simples utilizando TS.

[1] El archivo .c es el código fuente de entrada original, mientras que el archivo .i es el resultado del preprocesamiento de este código fuente.

Formato de errores semánticos <u>mínimos</u>:

- **Control de tipos de datos** simples de la operación binaria * (multiplicación)

    1. Cuando uno o ambos operandos del operador binario * tienen tipos de datos inválidos. ***L:C*** indica la ubicación del operador binario * en el archivo de entrada.

```
L:C: Operandos invalidos del operador binario * (tienen 'TIPO_DATO_L'y 'TIPO_DATO_R')
```

- **Control de declaración de símbolos**

    1. Cuando se utiliza un identificador con símbolo sin declarar en una expresión. ***L:C*** indica la ubicación del identificador en el archivo de entrada.

```
L:C: 'IDENTIFICADOR' sin declarar
```

    2. Cuando se redeclara un identificador con tipo diferente de símbolo (ej. variable/función - función/variable). $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: 'IDENTIFICADOR' redeclarado como un tipo diferente de simbolo
Nota: la declaracion previa de 'IDENTIFICADOR' es de tipo 'TIPO_DATO': L_B:C_B
```

    3. Cuando se redeclara/redefine un identificador con tipo igual de símbolo (ej. variable/variable - función/función) pero con tipos de datos diferentes. $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: conflicto de tipos para 'IDENTIFICADOR'; la ultima es de tipo 'TIPO_DATO_A'
Nota: la declaracion previa de 'IDENTIFICADOR' es de tipo 'TIPO_DATO_B': L_B:C_B
```

    4. Cuando se redefine una variable con tipos de datos iguales. $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: Redeclaracion de 'IDENTIFICADOR'
Nota: la declaracion previa de 'IDENTIFICADOR' es de tipo 'TIPO_DATO': L_B:C_B
```

    5. Cuando se redefine una función con tipos de datos iguales. $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: Redefinicion de 'IDENTIFICADOR'
Nota: la definicion previa de 'IDENTIFICADOR' es de tipo 'TIPO_DATO': L_B:C_B
```

- **Validación de invocación a funciones**

    1. Cuando el identificador de la invocación a función no se corresponde con ningún símbolo de la TS. ***L:C*** indica la ubicación del identificador en el archivo de entrada.

```
L:C: Funcion 'IDENTIFICADOR' sin declarar
```

    2. Cuando se trata de invocar a un objeto (ej. identificador de función) que no corresponde a ningún símbolo de la TS de tipo función o puntero a función. $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: El objeto invocado 'IDENTIFICADOR' no es una funcion o un puntero a una
funcion
Nota: declarado aqui: L_B:C_B
```

3. Cuando se pasan menos argumentos a una función de los que ésta requiere. $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: Insuficientes argumentos para la funcion 'IDENTIFICADOR'
Nota: declarado aqui: L_B:C_B
```

4. Cuando se pasan más argumentos a una función de los que ésta requiere. $L_A{:}C_A$ y $L_B{:}C_B$ indican las ubicaciones del identificador en el archivo de entrada.

```
L_A:C_A: Demasiados argumentos para la funcion 'IDENTIFICADOR'
Nota: declarado aqui: L_B:C_B
```

5. Cuando algún argumento no es compatible con el tipo de dato del parámetro de la función. $L_A{:}C_A$ indica la ubicación del argumento y $L_B{:}C_B$ la del parámetro de la función en el archivo de entrada.

```
L_A:C_A: Incompatibilidad de tipos para el argumento # de 'IDENTIFICADOR'
Nota: se esperaba 'TIPO_DATO_A' pero el argumento es de tipo 'TIPO_DATO_B': L_B:C_B
```

6. Cuando se trata de operar con el valor de retorno de una función que retorna *void* (ya sea para asignación, multiplicación, etc.). ***L:C*** indica la ubicación de la invocación de función en el archivo de entrada.

```
L:C: No se ignora el valor de retorno void como deberia ser
```

- **Validación de asignación**

1. Cuando se trata de inicializar una variable con un tipo de dato no compatible. ***L:C*** indica la ubicación del inicializador en el archivo de entrada.

```
L:C: Incompatibilidad de tipos al inicializar el tipo 'TIPO_DATO_L' usando el tipo
'TIPO_DATO_R'
```

2. Cuando se trata de reasignar el valor de una variable con calificador de tipo *const*. ***L:C*** indica la ubicación del operador de asignación en el archivo de entrada.

```
L:C: Asignacion de la variable de solo lectura 'IDENTIFICADOR'
```

3. Cuando no hay un valor-L modificable en la asignación. ***L:C*** indica la ubicación del operador de asignación en el archivo de entrada.

```
L:C: Se requiere un valor-L modificable como operando izquierdo de la asignacion
```

- **Validación de sentencias de salto *return***

1. Cuando una sentencia de salto return no retorna ningún valor en una función que retorna no *void*. ***L:C*** indica la ubicación del token *return* en el archivo de entrada.

```
L_A:C_A: 'return' sin valor en una funcion que no retorna void
Nota: declarado aqui: L_B:C_B
```

2. Cuando el tipo de dato de la sentencia de salto *return* no coincide con el tipo de dato de retorno de la función declarado. ***L:C*** indica la ubicación del valor de la sentencia de salto *return*.

```
L:C: Incompatibilidad de tipos al retornar el tipo 'TIPO_DATO_return' pero se esperaba
'TIPO_DATO_declarado'
```

Para simplificarse, puede considerarse que todos los símbolos (variables, funciones) están en el mismo alcance (scope) global y namespace. No afecta a los tests mínimos esperados. Pero en dado caso de que quiera hacer un trabajo más completo, el criterio es:

- En la TS se almacena el alcance en donde fue declarado cada símbolo.
- Dos o más símbolos pueden tener el mismo identificador sin haber colisión de símbolos si su alcance o namespace es distinto.
- Los símbolos de menor alcance tienen mayor precedencia cuando son referenciados por su identificador.
- El valor del alcance inicial es 0 (global). A mayor valor, menor alcance.
- Por cada llave de apertura (*{*) y llave de cierre (*}*) sintácticamente válidos se incrementa/decrementa su valor en 1, respectivamente.
- Al decrementar el valor, los símbolos que hayan sido declarados con ese alcance ya no podrán ser alcanzables en los niveles subsiguientes.

Criterio para manejo de errores:

- **Ante un error léxico**, capturar la secuencia de caracteres no reconocida. En este caso, no le llegará ningún token al analizador sintáctico. Pero puede suceder que un error léxico, desencadene un error sintáctico en caso de que el Parser no reciba los tokens adecuados para la GIC implementada. En general se debe retomar el análisis a partir del siguiente salto de línea '\n' o punto y coma ''; desde el analizador sintáctico.
- **Ante un error sintáctico**, utilizar el token "error" para capturar el error y almacenar la información que consideren importante para registrar el mismo. Luego seguir procesando a partir de donde encuentran un punto y coma ';' , o bien un salto de línea '\n'.
- **Ante un error semántico**, no es necesario salvar el error dado que la gramática independiente de contexto seguirá operando sin problemas. Pero deben implementar dentro de las rutinas semánticas la impresión de algún mensaje en pantalla, o bien, almacenar los errores encontrados en una lista para imprimirlo al final del análisis en el reporte.

El reporte deberá consistir en un conjunto de listados que deberá seguir el siguiente orden:

1. **Listado de variables declaradas** por orden de aparición indicando tipo de dato correspondiente a cada una. En caso de más de una variable declarada en la misma línea, respetar el orden de izquierda a derecha. No considerar el alcance (scope) de cada variable. *(Ya implementado en el TP3: utilizar TS)*
2. **Listado de funciones declaradas y definidas** por orden de aparición indicando para cada función el listado de parámetros de entrada (tipo, nombre) y el tipo de dato del valor de retorno. *(Ya implementado en el TP3: utilizar TS)*
3. **Listado de errores semánticos** encontrados por orden de aparición indicando en todos los casos el motivo, el número de línea y de columna correspondiente. Más abajo se detalla con un ejemplo cómo mostrar el listado. (*implementar con rutinas semánticas y TS*)
4. **Listado de errores sintácticos** encontrados indicando las estructuras que no sean derivables con el número de línea donde se encontraron. *(Implementar en Bison, utilizar token error)*
5. **Listado de errores léxicos** encontrados indicando las cadenas no reconocidas y el número de línea y el de columna donde se encontraron. *(implementado en el TP2 con Flex)*

Cuando no hay elemento de un listado se deberá imprimir en el reporte con un "-". Ver ejemplos de test incluídos en la carpeta correspondiente al trabajo práctico.

Uso

./bin/tp4.exe <ruta archivo entrada> [parámetros adicionales opcionales]

La ruta al archivo de texto de entrada es el único parámetro **obligatorio**.
Adicionalmente, pueden agregar parámetros **opcionales** que consideren útiles o convenientes (ej. activar logs verbosos para facilitar el debug, etc.).

Ejemplo de aplicación

Archivo de entrada (*archivo.i*) de prueba (no hace falta que compile para que lo tome):

```
void imprimir(void) {
    printf("Log: %s \n", "");
    return;
}

void imprimir(void);
int imprimir = 2, compartida = -5;
float potencia(float base, unsigned long);
int incremento(int y);
long incremento(long y);

int main(void) {
    unsigned int a = 06, b = "Hola mundo! \n", c = 0xA, c, d = c * "";
    const float e = potencia(1.0 * 2.0, 2 * 3) * (1.0 * 1), a, f = imprimir;
    if (a <= 6 && c != 'A') {
        int i;
        f(x);
        while (i > 0) {
            c = incremento(c);
            --i;
        }
    }

    switch (c) {
        case 10:
            c = a
            break;
        default:
            1 = 1;
            e @double = 0;
            return imprimir;
    }
```

```
    incremento();
    potencia(a * 1.0, a * c, 3 * imprimir);
    potencia(c * 3, e * 1.0);
    potencia(3.0 * f, e * f);
    potencia("++", incremento);
    a = imprimir();
    return "Terminado";
}

int incremento(int x) {
    return ++x;
}

int incremento(int x) {
    do {
        x+=1;
        continue;
    } while (x < 5);

    if (x > 5)
        if (x == 6 || x >= 10)
            return ++x;
        else {
            x += 2;
        }
    return x;
}

double potencia(double base, unsigned int exp) {
    float acumulador = 1.0;
    for (; exp > 0; exp--) {
        acumulador *= base;
    }
    return acumulador;
}
```

Ejecución:

./bin/tp4.exe archivo.i > salida.txt [2]

Archivo de salida (salida.txt) esperado:

```
* Listado de variables declaradas (tipo de dato y numero de linea):
compartida: int, linea 7, columna 19
a: unsigned int, linea 13, columna 18
b: unsigned int, linea 13, columna 26
c: unsigned int, linea 13, columna 48
d: unsigned int, linea 13, columna 60
e: const float, linea 14, columna 17
f: const float, linea 14, columna 64
i: int, linea 16, columna 13
acumulador: float, linea 63, columna 11

* Listado de funciones declaradas y definidas:
imprimir: definicion, input: void, retorna: void, linea 1
potencia: declaracion, input: float base, unsigned long, retorna: float, linea 8
incremento: declaracion, input: int y, retorna: int, linea 9
main: definicion, input: void, retorna: int, linea 12
incremento: definicion, input: int x, retorna: int, linea 43

* Listado de errores semanticos:
2:5: Funcion 'printf' sin declarar
7:5: 'imprimir' redeclarado como un tipo diferente de simbolo
Nota: la declaracion previa de 'imprimir' es de tipo 'void(void)': 1:6
10:6: conflicto de tipos para 'incremento'; la ultima es de tipo 'long(long)'
Nota: la declaracion previa de 'incremento' es de tipo 'int(int)': 9:5
13:30: Incompatibilidad de tipos al inicializar el tipo 'unsigned int' usando el
tipo 'char *'
13:57: Redeclaracion de 'c'
Nota: la declaracion previa de 'c' es de tipo 'unsigned int': 13:48
13:66: Operandos invalidos del operador binario * (tienen 'unsigned int' y 'char
*')
14:61: conflicto de tipos para 'a'; la ultima es de tipo 'const float'
Nota: la declaracion previa de 'a' es de tipo 'unsigned int': 13:18
14:68: Incompatibilidad de tipos al inicializar el tipo 'float' usando el tipo
'void (*)(void)'
17:11: 'x' sin declarar
17:9: El objeto invocado 'f' no es una funcion o un puntero a una funcion
Nota: declarado aqui: 14:64
```

[2] La extensión *.exe* sólo aplica para Windows. Éste comando redirige la salida por pantalla (stdout) del programa a un archivo (salida.txt). Si el archivo ya existe, lo sobreescribe; si no, lo crea.

```
29:15: Se requiere un valor-L modificable como operando izquierdo de la
asignacion
30:23: Asignacion de la variable de solo lectura 'e'
31:20: Incompatibilidad de tipos al retornar el tipo 'void (*)(void)' pero se
esperaba 'int'
34:5: Insuficientes argumentos para la funcion 'incremento'
Nota: declarado aqui: 9:5
35:32: Operandos invalidos del operador binario * (tienen 'int' y 'void
(*)(void)')
35:5: Demasiados argumentos para la funcion 'potencia'
Nota: declarado aqui: 8:7
38:14: Incompatibilidad de tipos para el argumento 1 de 'potencia'
Nota: se esperaba 'float' pero el argumento es de tipo 'char *': 8:22
38:20: Incompatibilidad de tipos para el argumento 2 de 'potencia'
Nota: se esperaba 'unsigned long' pero el argumento es de tipo 'int (*)(int)':
8:28
39:7: No se ignora el valor de retorno void como deberia ser
40:12: Incompatibilidad de tipos al retornar el tipo 'char *' pero se esperaba
'int'
47:5: Redefinicion de 'incremento'
Nota: la definicion previa de 'incremento' es de tipo 'int(int)': 43:5
62:8: conflicto de tipos para 'potencia'; la ultima es de tipo 'double(double,
unsigned int)'
Nota: la declaracion previa de 'potencia' es de tipo 'float(float, unsigned
long)': 8:7

* Listado de errores sintacticos:
"c = a": linea 26

* Listado de errores lexicos:
@double: linea 30, columna 15
```

Consideraciones

La entrega de este trabajo práctico **grupal** es **obligatoria y con exposición sincrónica**, en la **fecha y horario indicados en el cronograma**, acordado previamente con cada equipo. Deben estar presentes con cámara y micrófono todos los integrantes del equipo. La defensa tendrá una duración aproximada de 15 minutos en los cuales todos los integrantes deberán responder preguntas referidas al trabajo práctico integrador.

Las directivas del preprocesador y los comentarios no estarán presentes en el archivo *.i* que se recibe en esta etapa de la compilación puesto que se asume que todas ya han sido resueltos previamente en la etapa de preprocesamiento por el PREPROCESADOR. Es por eso que el analizador léxico correspondiente al COMPILADOR no los reconoce también – ya no hace falta en esta etapa.[3]

En el apunte *"SSL - Volumen 1" (Muchnik)* y en la *BNF Simplificada de C* tienen disponibles gramáticas de expresiones las cuales pueden utilizar para escribir algunas de las primeras reglas de *Bison* correspondientes a expresiones.

En el *Aula Virtual* también tienen a disposición **resúmenes de sintaxis del lenguaje ANSI C (*C89/C90*)** *(Language Syntax Summary)*, los cuales incluyen una gramática *(Grammar)* completa que puede ser de utilidad para escribir todas las reglas en *Bison* (tanto para expresiones, declaraciones como sentencias).

Dicha gramática está explicada producción a producción en *"El Lenguaje de Programación C" (2da edición, 1988) (Kernighan, Ritchie)*, así como en los estándares "*ANSI X3.159-1989*" y "*ANSI/ISO 9899-1990*" (también disponibles en el Aula Virtual), que pueden tomar como referencia para poder entender sus no-terminales, producciones y/o las restricciones sintácticas y semánticas que conllevan.

Basar el desarrollo del trabajo práctico en los ejemplos mostrados en clase que están disponibles **en el repositorio con ejemplos de Flex y Bison**.

Para la corrección y los tests se consideran únicamente las estructuras sintácticas mínimas requeridas. En caso de querer hacer un trabajo más completo, se pueden considerar tipos de dato puntero, arreglos, structs, unions, etc. Tener presente de todos modos que la declaración de variables simples puede incluir más de una por línea. No es necesario considerar el alcance (scope) de cada variable para poder superar los tests mínimos.

Recomendaciones:

- Ir de menos a más en la construcción de las gramáticas. Camino sugerido: 1) Expresiones 2) Declaraciones 3) Sentencias. Hacerlo de forma iterativa e ir probando. En varios casos pueden empezar implementando las ***producciones más sencillas*** y luego ir agregando complejidad.
- No copiar y pegar sin entender la gramática antes. Hay que ir integrando el código. En ningún caso copiar y pegar la gramática en un solo paso.

---

[3] En la práctica (*gcc* al menos), el PREPROCESADOR podría dejar algunas directivas de preprocesador sin expandir (por ejemplo, tras resolver un *#include*) que terminan quedando en el archivo *.i* que genera. Es por ello que los archivos *.i* generados por *gcc* no suelen servir como entrada para el TP, que es una versión simplificada y reducida de la realidad, la cual consiste en que el COMPILADOR también resuelve aquellas directivas que pudieran haber quedado para su etapa (e incluso tener que interpretar algún pseudocódigo que haya quedado tras resolverse una directiva *#line* por ejemplo). Como entrada para el TP, pueden utilizar los archivos *.i* de test y/o escribir los suyos.

- Cada TP suele ser una combinación única de reglas léxicas y sintácticas. Tener presente que la consulta con otros equipos puede ser enriquecedora pero no habrá posibilidad de trasladar el código de forma directa.

Para su realización, **se debe llevar un registro del número de línea y el de columna actual** en la que el analizador léxico se encuentra leyendo del archivo de entrada, e ir pasando las ubicaciones de los tokens al analizador sintáctico a través de la variable global *yylloc* (habilitado mediante el uso de la directiva *%locations* de *bison*).

Se recomienda **que el programa también tenga una o más opciones para poder depurarse** para facilitar el seguimiento y prueba del mismo.[4] [5] [6] Por ejemplo:

- Imprimir (o no) qué cadenas el analizador léxico va reconociendo y no reconociendo, a qué categoría léxica pertenecen, y el número de línea y de columna donde fueron encontradas.
- Imprimir (o no) por qué reglas de *Bison* el analizador sintáctico va reduciendo, y el número de línea y de columna a la que corresponden.

En este caso, se espera que la depuración pueda habilitarse/deshabilitarse a través de una o más variables (ej. *bool DEBUG*) y/o definiciones de preprocesador (ej. *#define DEBUG*) con un valor binario (*1* ó *0*) según necesidad del equipo, y que por la salida estándar (*stdout*) únicamente se imprima la salida esperada (el reporte), de forma que los tests automáticos no fallen. Para imprimir los mensajes de depuración puede entonces utilizarse la salida estándar para errores (*stderr*), la cual también se muestra por pantalla en combinación con *stdout*, pero sin afectar a los tests automáticos.

Para la generación del reporte es necesario **utilizar memoria dinámica** en la implementación del analizador sintáctico.

Para escribir el código fuente del programa pueden utilizar cualquier estándar de C, como: ANSI C (*C89/C90)*, *C99*, *C11*, *C17/C18*, etc.[7]

---

[4] Si a *flex* se le agrega la opción *-d* al producir el archivo *.lex.yy.c*, al compilar y ejecutar el analizador léxico generado se depura por qué reglas va entrando el mismo.

[5] Si a *bison* se le agrega la opción *-t* al producir el archivo *.tab.c* (y/o también a *gcc* se le agrega la opción *-DYYDEBUG=1* al preprocesar dicho archivo, por ejemplo), cuando se compile el analizador sintáctico también lo harán las herramientas de depuración de *bison*.

Además, para depurar el analizador sintáctico, la variable de tipo int *yydebug* debe tener asignado un valor entero distinto de 0 en tiempo de ejecución. Una manera sencilla de lograr esto es agregar el siguiente código antes de que se llame a *yyparse()*:

```
#if YYDEBUG
  yydebug = 1;
#endif
```

[6] Si a *bison* se le agrega la opción *-v* al producir el archivo *.tab.c*, se genera también un archivo *.output* que contiene información en texto plano sobre el analizador sintáctico generado.

Se puede indicar qué cosas más se reportan en dicho *.output* generado agregando también opciones como *--report=state* , *--report=itemset* , *--report=lookahead* (y las que estén disponibles según la versión instalada de bison, las cuales figuran en el manual correspondiente).

[7] En *gcc* el estándar de C utilizado se especifica agregando una opción *-std* al compilar, entre las que están: *-std=c89*, *-std=c99*, *-std=c11*, *-std=c17*, etc. Para conocer todas las opciones *-std* (ergo todos los estándares) que soporta una versión de *gcc*, pueden consultar la documentación de GCC que corresponda a dicha versión. Con *gcc --version* pueden consultar la versión instalada.

Si no se indica una opción *-std* al compilar, *gcc* por defecto utiliza el último estándar de C que soporte pero incluyendo las extensiones del lenguaje propias del compilador.

Como compilador de C se utilizará *gcc* (***G**NU **C C**ompiler*). Sin embargo, se recomienda **no utilizar** extensiones del lenguaje C específicas del compilador (*GNU C Extensions*) que no formen parte del estándar de C utilizado.[8]

El programa debe poder compilarse independientemente del sistema operativo, utilizando únicamente bibliotecas estándar de C según necesidad: *stdio.h*, *stdlib.h*, *ctype.h*, *string.h*, *math.h*, *errno.h*, etc.

En otras palabras, **el código C escrito debe ser portable**; para ello, no debe incluir bibliotecas específicas de ningún sistema (ej. *Windows*, *Linux*, *Mac OS*, etc.). Por ejemplo, bibliotecas definidas en los estándares *POSIX* (***P**ortable **O**perating **S**ystem **I**nterface/Uni**x***), ya que no son compatibles nativamente con Windows, entre otros sistemas.

Pueden utilizar cualquier versión de Flex y de Bison.[9] Como suelen ser retrocompatibles, solamente deberían consultar las documentaciones correspondientes a las versiones más antiguas de Flex y de Bison de las que tengan instaladas entre los integrantes del equipo, para que se aseguren de utilizar características que estén presentes en todas ellas y que de esa forma todos puedan construir el proyecto. Se sugiere la documentación de Flex 2.5 y Bison 2.4.1, las cuales están presentes en el Aula Virtual junto con la de las demás versiones.

El **entorno de desarrollo a utilizar queda a elección de cada grupo** (*Eclipse*, *CodeBlocks*, CLion, ***Visual Studio Code***, etc.). Se recomienda un entorno que aplique resaltado de sintaxis para todos los archivos de especificación, que facilite la depuración de código C, así como que se integre con la consola de *Git* para poder realizar el trabajo de una forma más práctica.

Para compilar el programa, pueden utilizar herramientas como ***GNU Make*** con archivos de especificación correspondientes como ***makefiles***, ya sean de su propia autoría o los que les hayan sido provistos (en el *aula virtual* o en las plantillas de TPs en los repositorios grupales designados). En dicho caso, **esos archivos deberán formar parte de la entrega**.

---

[8] *gcc* puede alertar del uso de extensiones de C como warnings si se le agrega la opción *-Wpedantic* y/o como errores si se le agregan las opciones *-pedantic -pedantic-errors* al compilar.

[9] Con *flex --version y bison --version* pueden consultar las versiones instaladas de Flex y Bison, correspondientemente.

Para el programa, sólo formarán parte de la entrega los **archivos de especificación** de Yacc/Bison (*.y*)**,** Lex/Flex (*.l*)**, archivos fuente** de C (*.c*) **y/o archivos de cabecera** (*header files*) (*.h*) escritos por el equipo. No se considerarán archivos generados por Yacc/Bison (ej. *.tab.c* , *.tab.h* , *.output*), Lex/Flex (ej. *.lex.yy.c*), ejecutables (ej. *.exe*) ni archivos intermedios de salida del compilador, como los archivos objeto (ej. *.o*), ensamblador (ej. *.s*) o preprocesados (ej. *.i*).

Completar el archivo *README.md* dentro del directorio del TP, el cual actúa como carátula del mismo, con los datos que allí consignan.

La **entrega será a través del repositorio** de GitHub designado para el equipo en la carpeta correspondiente al TP **mediante Pull Request (PR)** desde la rama creada para el TP hacia la rama principal (main), como se indica en el instructivo para entrega de trabajos prácticos.

Pueden **agregar también las aclaraciones que consideren necesarias mencionar** en relación con el trabajo práctico realizado **en dicho Pull Request (PR)**.

Las **consultas** podrán ser realizadas a través de Discord en el canal correspondiente a cada grupo, o bien, si es una consulta más general del trabajo referida al enunciado por el canal de consultas denominado TP-**#**, siendo **#** el número de TP.