UTN - FRBA
Sintaxis y Semántica de los Lenguajes
Trabajo Práctico n° 3 - *Bison* para reconocimiento de estructuras sintácticas

Enunciado

Implementar en lenguaje C un programa que permita reconocer a partir de la entrada de un archivo fuente *.i* (archivo *.c* preprocesado)[1] las **categorías sintácticas 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* (en conjunto con *Flex*) para la generación del código C que implemente un analizador sintáctico (*parser*) que cumpla con lo requerido.

Categorías sintácticas de ANSI C mínimas a reconocer:

- **Expresiones**
  - Primarias: *<identificador>*, *<constante>*, *<literal cadena>* y *(<expresión>)*
  - 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 ||
  - De asignación: operadores = , += , -= , *= y */=*
- **Declaraciones**
  - Variables (simples)
  - Prototipos de funciones (simples)
- **Sentencias**
  - De expresión
  - Compuestas
  - De selección: *if* , *if/else* y *switch*
  - De iteración: *while* , *do/while* y *for*
  - Etiquetadas: *case* y *default*
  - De salto: *continue* , *break* y *return*
- **Definiciones externas** (*globales*)
  - Definiciones de funciones (simples)
  - Declaraciones (*globales*)

[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.

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'.

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.
2. **Listado de funciones declaradas/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.
3. **Listado de tipos de sentencias** por orden de aparición indicando en todos los casos el tipo de sentencias, el número de línea y de columna correspondiente. Más abajo se detalla con un ejemplo cómo mostrar el listado.
4. **Listado de estructuras que no sean válidas sintácticamente** indicando el número de línea donde se encontraron (indicar las líneas donde se encontraron errores sintácticos).
5. **Listado de cadenas no reconocidas** indicando el número de línea y el de columna donde se encontraron (esto ya lo tienen implementado en el TP 2)

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/tp3.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):

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

j = a;

void rutina(int x);

int main(void) {
    unsigned int i, j = 0xA, a = 06;
    if (a > 5 && j == 10) {
        int b = a;
        while (b != 0) {
            printf("El valor de b es %d\n", b);
            b--;
            ++j;
            continue;
        }
    }

    switch(potencia(j, 2)) {
        case 2:
            break;
        case 4:
            e @double = 0;
            return b-j;
        default:
            return 1+j+b;
    }

    rutina(5);
    return 0;
}
```

```c
void rutina(int x) {
    do {
        printf("El valor de x es %d \n", x);
        x+=1;
    } while (x < 5);

    if (x == 5 || x > 10) {
        printf("x es 5 o mayor que 10 \n");
    } else {
        printf("x tiene un valor entre 6 y 9 inclusive \n");
    }
}
```

Ejecución:

./bin/tp3.exe archivo.i > salida.txt [2]

Archivo de salida (salida.txt) esperado:

```
* Listado de variables declaradas (tipo de dato y numero de linea):
acumulador: float, linea 2
i: unsigned int, linea 14
j: unsigned int, linea 14
a: unsigned int, linea 14
b: int, linea 16

* Listado de funciones declaradas o definidas:
potencia: definicion, input: float base, long exp, retorna: float, linea 1
rutina: declaracion, input: int x, retorna: void, linea 11
main: definicion, input: void, retorna: int, linea 13
rutina: definicion, input: int x, retorna: void, linea 39

* Listado de sentencias indicando tipo, numero de linea y de columna:
for: linea 3, columna 5
return: linea 6, columna 5
if: linea 15, columna 5
while: linea 17, columna 9
continue: linea 21, columna 13
switch: linea 25, columna 5
case: linea 26, columna 9
break: linea 27, columna 13
case: linea 28, columna 9
return: linea 30, columna 13
default: linea 31, columna 9
return: linea 32, columna 13
return: linea 36, columna 5
do/while: linea 40, columna 5
if/else: linea 45, columna 5

* Listado de estructuras sintacticas no reconocidas:
"j = a;": linea 9

* Listado de cadenas no reconocidas:
@double: linea 29, columna 15
```

[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.

Consideraciones

La entrega de este trabajo práctico **grupal** es **obligatoria**, su **fecha límite para consulta, entrega y revisión se encuentra indicada en el cronograma**. Luego de esa fecha, **no** se aceptarán más trabajos, y toda consulta referida quedará para la defensa final que será al cierre de la cursada.

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.

---

[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.

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.
- 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 implementacióndel analizador sintáctico.

---

[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).

El agrupamiento por tipo de las palabras reservadas utilizado en el TP 2 puede utilizarse como base pero deberá separarse en una regla por cada palabra reservada, dado que a nivel sintáctico pueden involucrar distintos tipos de producciones en la BNF.

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

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.

---

[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.

[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 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**.

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.