Práctica 3

Un analizador sintáctico descendente para "miLenguaje"

1. Objetivos

1. Desarrollar un analizador sintáctico descendente para un lenguaje imperativo
2. Integrar un analizador léxico con un analizador sintáctico
3. Usar la utilidad "make" de Unix

2. Contenidos

Como objetivo final, esta práctica propone el desarrollo de un analizador sintáctico descendente para "miLenguaje". Como primer paso, es necesario escribir la gramática del lenguaje que habéis especificado en la primera práctica. Es aconsejable, primero, definir una gramática "intuitiva" para, en un paso posterior, tratar de transformarla en una que sea LL(1). Una gramática para un lenguaje procedural tipo Ada puede empezar como:

```
programa ->
  tkPROCEDURE
  tkID
  listaParsFormales
  tkIS
  parteDecs
  bloqueInst
```

```
  ';'

listaParsFormales ->
  | listaParsF

listaParsF ->
  listaParsF ';' decPar
  | decPar

. . .
```

Es importante que antes de empezar a implementar el analizador meditéis concienzudamente sobre la gramática y cualquier cuestión la comentéis con el profesor de prácticas.

3. Resultados

Como resultado de esta práctica hay que entregar el fichero denominado `pract3.tar`, de manera que la ejecución de la instrucción `tar xvf pract3.tar` genere un directorio denominado `pract3` con los siguientes ficheros:

- `AL_miLenguaje_2.l` que contiene el fuente Flex del analizador léxico usado (notar que el nombre no puede coincidir con el que sometiériais para la segunda práctica, por lo que ha sido cambiado).
- `AS_LL1.c` que contiene el fuente C con la implementación del analizador sintáctico. La implementación puede realizarse tanto por tabla como descendente recursivo.
- `AS_LL1Make` que contiene el fichero Make para generar el ejecutable del analizador sintáctico, de manera que la invocación
  `make -f AS_LL1Make`
  genere el ejecutable `AS_LL1` que realiza el análisis sintáctico. La invocación al analizador debe ser como sigue (el significado del número 3 en la invocación se explicará más tarde):

  ```
  AS_LL1 3 nombreDelFicheroConElFuente
  ```

- Todos aquellos ficheros que sean necesarios (conteniendo, por ejemplo, implementaciones de tipos de datos, funciones para el tratamiento de errores, etc.).

El analizador léxico no debe dar ningún mensaje (ni realizar ninguna tarea adicional) cuando el fuente sea léxica y sintácticamente correcto.

En caso de error léxico, el comportamiento ha de ser el establecido en la segunda práctica. En caso de error sintáctico, se debe suministrar información al usuario como se muestra en el siguiente ejemplo. Considerar, como ejemplo, el programa en Ada

```
Procedure is
  a, c: integer;
Begin
  get(a);
  If a>4 Then
    c:=a;
  End If;
  put("Se acabó"); new_line;
End;
```

Deberá dar un mensaje de error del estilo del siguiente:

```
(1) Error:lin_1: Se esperaba un identificador
Procedure is ....;
----------^
```

La sintaxis que aquí se utiliza para la salida de errores sintácticos es:

- entre paréntesis el número identificativo que el analizador asigna al error; empieza por 1 y se incrementa correlativamente (de momento, pararemos en cuanto encontremos uno; cuando implementemos recuperación de errores, podrán detectarse varios en una misma pasada).
- a continuación `lin_#`, donde `#` representa el número de la línea donde se ha detectado el error.
- después información referente al error detectado. Esta información debe ser tan precisa y útil para el usuario como sea posible

- a continuación, en la línea siguiente, el texto que se lleva reconocido de la línea que se está procesando
- en la línea siguiente, un apuntador a la posición precisa donde se ha localizado el error.

Es preciso que tengáis en cuenta que para que el último apartado sea viable, es necesario saber a cuántos espacios en blanco ha de equivaler un tabulador cuando se escribe el mensaje de error. Para ello, la invocación a la aplicación será como sigue:

```
AS_LL1 3 nombreDelFicheroConElFuente
```

donde `3` indica el número de espacios a que equivale un tabulador (habitualmente, la salida estándar está configurada, por defecto, a 8 espacios en blanco por tabulador).

Notar que será necesario que hagáis una nueva versión del analizador léxico para poder ir almacenando el texto de la línea que se está procesando de acuerdo con las nuevas especificaciones. Suponer para ello que ningún programador con sentido común utilizaría nunca una línea de más de 132 caracteres.

4. Observaciones

Para establecer la gramática de miLenguaje será de ayuda consultar las que aparecen en los libros de C, Ada, Pascal, etc. Suele ser especialmente crítica la parte de la gramática correspondiente a las expresiones, por lo que es importante que esta parte se trate con cuidado.

No construyáis toda la gramática de una sola vez. Probar a hacerlo de una manera incremental. Construir primero la gramática que sólo reconozca la declararación del procedimiento/función y su cuerpo. Progresivamente, ir añadiendo nuevas reglas, realizando tests, y así sucesivamente. La gramática completa de un lenguaje de las características del que estamos manejando ocupa alrededor de tres páginas de fuente Yacc. Su transformación para el análisis LL(1) ocupará un poco más (pero no mucho más).

5. Entrega de la práctica

Obligatoriamente, cada grupo debe tener terminada la tercera práctica de manera que pueda ser revisada y comentada con el profesor de prácticas durante la quinta sesión de prácticas.

6. Anexo

Ejemplo de fuente para "make"

```
#=========================================================
# Fich.:  AS_miniAdaMake
# Tema:   Fichero Make para construir un an. sintáctico
#         para miniAda. Usa el an. léxico 'AL_miniAda_2.l'
#         El AS se genera a partir de 'AS_miniAda.c'
# Fecha:  Octubre-03
# Fuente: JE
# Com.:   Para las prácticas de COMP-I. C.P.S.  UniZar
# OJO:    puede que, de acuerdo con el
#         enunciado concreto, cambien los nombres usados
#=========================================================
LEX=flex
CC=gcc
SALIDAS=/users2/COMPI/salidas
LIBS=/opt/flex/lib

AS_miniAda: lex.yy.o AS_miniAda.o
    $(CC) -o AS_miniAda -I$(SALIDAS) -L$(LIBS) \
        AS_miniPascal.o lex.yy.o -lfl

lex.yy.o: lex.yy.c
    $(CC) -c -I$(SALIDAS) lex.yy.c

AS_miniAda.o: tokens.h AS_miniAda.c
    $(CC) -c AS_miniAda.c

lex.yy.c: AL_miniAda_2.l tokens.h
    $(LEX) AL_miniAda_2.l
```