Práctica 1: Construcción de un analizador léxico para *adac*

Procesadores de lenguajes

Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivos

Los objetivos para esta práctica son:

- Conocer las características del lenguaje de programación *adac*, que manejaremos a lo largo de las sesiones de laboratorio
- Aprender las características y funcionamiento del metacompilador JavaCC
- Implementar un analizador léxico para *adac*, incluyendo alguna ténica de recuperación de errores léxicos

2. Caracerísticas de *adac*

En esta práctica deberás desarrollar un analizador léxico *adac*. *adac* es un lenguaje estructurado sencillo, con algunas características similares a Ada, y otras a C, entre las cuales destacan:

1. Los comentarios empiezan con la marca `--` (doble guión) y terminan al terminar la línea.
2. Se permite la declaración y uso de variables simples (escalares), tanto globales como locales, de tres tipos: entero, carácter y booleano. La declaración de variables sigue una sintaxis parecida a la de C:

```
integer i, j, k;
character c, d, e;
boolean v, f;
integer z;
```

3. Ningún símbolo puede llamarse como las palabras reservadas. Es decir, no se permite declarar una variable o una función de nombre "integer", o "if", por ejemplo.

4. Los identificadores estarán compuestos por letras, números y el símbolo "_" (carácter *underscore*), con las siguientes restricciones:
   - Un identificador nunca puede comenzar por un número.
   - No se admiten dos caracteres "_" seguidos.

5. Las palabras reservadas se han de escribir en minúsculas.

6. El lenguaje es **case-sensitive**. Es decir, distingue mayúsculas de minúsculas. El identificador "X" es distinto de "x". Aunque no es aconsejable su uso, nada impide que se utilice "IF" como nombre de variable.

7. Las constantes de tipo carácter se escriben entre comillas simples. El carácter comilla simple se indica mediante tres comillas simples:

```
c := 'a';
c := '"';
c := ''';
```

8. No está permitida la declaración de procedimientos y funciones anidados.

9. Cuando un procedimiento o función no tiene parámetros, debe invocarse con paréntesis vacíos, "()".

10. Hay dos formas de paso de parámetros: por valor y por referencia, con la semántica habitual. Se marcan con las palabras clave **val** y **ref**, respectivamente. Es obligatorio prefijar la forma de paso de cada parámetro (clase del parámetro).

```
procedure procesar(val integer i, j, k; ref character c, d, e;
                   ref boolean f) is ...
```

11. Se permite la escritura de variables y expresiones simples (no de "arrays"), mediante las instrucciones *put* y *put_line* (esta última añade un salto de línea tras la escritura). Como salida, la operación de escritura mostrará por la salida estándar el valor correspondiente a las expresiones de tipo entero o carácter, y las cadenas **true** o **false** en el caso de booleanos. También se permite la escritura de constantes de tipo cadena.

12. Se permite el uso de cadenas de caracteres constantes, aunque solamente para escritura:

```
put_line("x:␣",x);
```

Las constantes string no usan secuencias de escape. Cuando dentro de un string se desee usar el carácter ", debe ponerse doble. Así, si se ejecutara

```
put_line("Hola␣""␣caracola");
```

el resultado debería ser

```
Hola " caracola
```

Cuando se necesiten caracteres especiales que requieran habitualmente estar "escapados", se deberán usar las funciones `int2char` y `char2int`. Así, si consideramos el siguiente código C++

```
char c;
c = '\n';
cout << "Hola\tCaracola\n";
```

podríamos escribir en *adac* uno que se comporta de la misma manera como sigue (aunque hay muchas alternativa posibles, claro):

```
character c;
c := int2char(10);
put("Hola",int2char(9),"Caracola",c)
```

13. El lenguaje dispone de funciones que devuelven datos escalares (**character**, **integer**, **boolean**). La devolución se lleva a cabo mediante la instrucción "return". La sintaxis es la que se muestra a continuación:

```
function integer valAbs(val integer x) is
begin
    if x < 0 then
        return -x;
    else
        return x;
    end
end
```

14. El lenguaje maneja también arrays, que se declaran con una sintaxis parecida a la de C:

```
integer v1[100],v2[100];
....
```

```
procedure datos (val integer d; val integer w[100]) is
    integer v[100];
begin
    ...
    v[3] := w[9];
    ...
end
```

Veamos un ejemplo de programa completo en *adac* (consúltese la batería de programas suministrada para tener una visión completa de todos los elementos del lenguaje):

```
procedure mcd is --el proc principal no toma parámetros
integer i, j;
-------------------------------------------------------
procedure dato(ref integer d) is
begin
    put("Num␣(>0):␣");
    get(d);
    while d <= 0 do
        put_line("El␣numero␣debe␣ser␣positivo.");
        put("Num␣(>0):␣");
        get(d);
    end
end
-------------------------------------------------------
function integer mcd(val integer a; val integer b) is
    integer aux;
begin
    while b <> 0 do
        aux := a;
        a := b;
        b := aux mod b;
    end
    return a;
end
-------------------------------------------------------
begin
    dato(i);
    dato(j);
    put("MCD(",i,",",j,")=␣");
    put_line(mcd(i,j));
end
```

3. Tareas a realizar

Obligatoriamente, hay que realizar las siguientes tareas:

1. Construir un analizador léxico para *adac*. El fichero fuente a analizar se suministra como un parámetro en la invocación desde la línea de comandos. Para cada token reconocido, debe mostrar por la salida estándar la información correspondiente, de la siguiente manera, donde el par entre paréntesis corresponde a la línea y columna donde se ha localizado el lexema correspondiente al token:

```
(25,73):  identificador "edad"
(170,333): operador asignación ":="
(4,90):  constante entera "899"
...
```

2. En el caso de detectarse un error léxico, deberá mostrar por la salida estándar un mensaje del tipo:

```
ERROR LÉXICO: (<línea, columna>): símbolo no reconocido: <símbolo>
```

3. Escribir 5 programas de test en *adac*, que complementen los que se suministran como batería de test. Estos deben resolver problemas interesantes.

Algunos comentarios:

- Recordad que en esta práctica sólo hay que hacer un análisis léxico, no importa si la estructura del programa es correcta o no. De eso se encargarán fases posteriores del análisis.
- De los programas de prueba que se os proporcionan, todos son correctos desde el punto de vista léxico.

4. Entrega de resultados de la práctica

4.1. Lo que hay que entregar

Como resultado de la práctica, todos los alumnos deberán entregar el fichero `practica_1.zip`. Este, una vez descomprimido, tendrá la estructura que se muestra en la parte izquierda de la figura 1. Los nombres de los ficheros y directorios mostrados deben respetarse. La estructura corresponde a lo siguiente:

- El fichero `adac.jj` contiene el fuente del analizador léxico pedido
- El fichero `README.txt` contiene información sobre los autores: nombres, NIPs, etc.
- El directorio `lib` de momento está vacío, pero en un futuro contendrá librerías que desarrollemos y se necesiten para la compilación del procesador. Por ejemplo, librerías para el manejo de la tabla de símbolos, de procesamiento semántico, etc.

- El fichero `build.xml` contiene la descripción del proyecto. Este fichero es el que se suministra en la web de la asignatura (o una adaptación del mismo cuando sea necesario). El funcionamiento de "ant" es análogo al de un "makefile". En caso de invocaciones sucesivas, solo compilará los ficheros que hayan sufrido cambios desde la última vez que se invocó, y los que dependan de ellos (buscar información al respecto). Para adaptarlo a vuestro entorno, basta con modificar la línea siguiente con el path hasta donde hayáis instalado javacc en vuestra máquina.

```
javacchome="${user.home}/aplicaciones/javacc/target"
```

La ejecución del comando `ant` llevará a cabo la compilación del proyecto. Si todo ha ido correctamente, tras su ejecución, el contenido del directorio será el mostrado en la parte derecha de la figura 1. En el nuevo estado:

- Se ha creado el directorio `jcc_files` con los fuentes java generados por `javacc`, que implementan el analizador léxico especificado en `adac.jj`
- Se ha creado el directorio `build`, que contiene las clases compiladas, así como otra información especificada en el proyecto.
- Se ha creado el directorio `dist` con los materiales para la distribución de la aplicación. Contiene, por un lado, el directorio `doc`, que es una copia del original. Por otra parte, la aplicación generada, `adac.jar`. Para analizar el fuente `mcd.adac`, la instrucción a ejecutar será

```
java -jar adac.jar mcd.adac
```

Por último, la ejecución del comando `ant clean` devolverá el directorio a su estado original, eliminando todo lo que hubiera generado (incluyendo el directorio de distribución, claro).

4.2. Método de entrega

La entrega se debe hacer en *hendrix* mediante la ejecución del programa

```
someter procleng_21 practica_1.zip
```

Nota: en la entrega de esta práctica no hay incluir los 5 programas de prueba que se nombran en la sección 3. Estos deberán ir en la entrega de la práctica 2.

4.3. Plazos de entrega

El plazo de entrega es de dos semanas, a contar desde el día de la realización de la práctica. Quien tenga la sesión en martes, deberá entregarla antes del segundo martes después de su realización. Análogamente para el resto de días de prácticas.

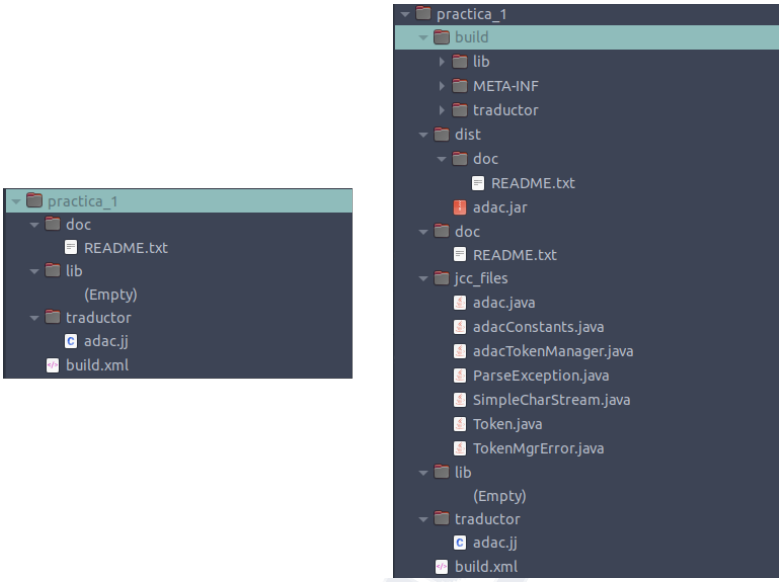

Figura 1: Estructura del proyecto antes y después de su compilación