# Procesador de Lenguaje – Parte 1

Antonio Manuel Guisado Valle

# 1. El analizador léxico

En esta parte de la práctica desarrolle la captura de los diferentes tokens necesarios para el lenguaje Ada Uned. Estos tokens están sacados en base a este lenguaje. Solo comentar varias cosas.

Lo primero es que el orden en el que aparecen las diferentes expresiones regulares que capturan los tokens no es al azar. Se tienen que poner de mas restrictivo a menos porque, si lo pusiéramos de la otra manera, todos los patrones encajarían en la expresión menos restrictiva. Por ejemplo, si pusiéramos la expresión regular para definir el token IDENTIFICADOR al principio de la sección YYINITIAL, ninguna palabra clave o cadena podría identificarse porque todo se vería como tokens IDENTIFICADOR.

Además, estos tokens los he definido estudiando como y donde deben aparecer en el lenguaje AdaUned y, basándome en esto, escribir una expresión regular que lo capture.

También comentar la inclusión de las diferentes configuraciones necesarias para el buen funcionamiento del programa. Empezamos por “%ignorecase”, necesaria para la correcta lectura del programa y para que no diferencie entre mayúsculas y minúsculas. Además, añadí la configuración “%notunix” para que reconociera tanto \r\n y \n como nueva línea, así facilitar también la lectura de los tokens, en especial el de los comentarios para que al leer \n no saltara línea al imprimir la pantalla por consola. Por último, añadí la configuración “%unicode” , con la que podemos reconocer tanto caracteres especiales, como la ñ, o caracteres acentuados. Por otro lado, cree tres funciones diferentes para agilizar el proceso de programación de esta parte de la práctica. La primera función que vamos a comentar es “crearToken”, ya que si no creaba esta función era copiar y pegar las mismas sentencias en casi todos los tokens para crearlos y enviárselos al analizador sintactico y, al haberlo hecho función, queda un código mucho mas limpio y corto. También comentar la función “crearCadena”. Esta función cumple el mismo propósito que “crearToken” pero exclusivamente para el token cadena, ya que esta función esta modificada con el comando substring para eliminar tanto el primer como el ultimo elemento de la cadena, o lo que viene a ser lo mismo, eliminar las comillas de las cadenas y poder enviarlas al sintáctico sin ellas. Por último, tenemos la función “crearError”, para hacer lo mismo que la función “crearToken” pero para detectar posibles errores.

A continuación, comentar la creación de macros para facilitar la escritura de las expresiones regulares. Solo realice dos diferentes, una para identificar cualquier letra del abecedario llamada “letra”, y la otra para facilitar la labor de poner los diferentes espacios, saltos de línea y demás en una sola expresión y es “ESPACIO_BLANCO”.

Por último, comentar que los diferentes tokens, además de definirlos en el archivo “scanner.flex” , hay que definirlos en el archivo “parser.cup”, encargado del analizador sintáctico, como terminal token y así conectar ambos archivos.

## 2. Analizador sintáctico

En este segundo apartado tenemos el analizador sintáctico, el cual desarrolle teniendo en cuenta lo hecho en el analizador léxico y, así, conseguir que reconozca el lenguaje propuesto.

La declaración de los terminales usados en este apartado ya la comentamos en el apartado anterior. La declaración de los no terminales la fui desarrollando mientras iba construyendo las diferentes reglas de producción.

Las reglas de precedencia las saque directamente del enunciado de la práctica. Estas reglas son sumamente importantes para evitar la ambigüedad.

Pasando a las reglas de producción, no seguí para su creación ningún principio específico, simplemente fui creando las diferentes construcciones por separado y, después, las uní para que formaran un uno. Por ejemplo, primero definí los distintos tipos de declaraciones y hasta que no las tenía todas creadas no las unifique y solucione los posibles problemas y ambigüedades que eso conllevaba.

Ya que hablamos de los tipos de declaraciones (constantes simbólicas, tipos globales, variables y subprogramas), al unirlas e intentar que estuvieran en orden como deben de estar en el lenguaje AdaUned, me daban ambigüedades y conflictos por todos lados. El principal problema es que tanto la declaración de las constantes, como de los tipos primitivos como el de las variables comparten una cosa, que empiezan con un identificador. Esto hace que el analizador no sepa ni si lee uno a cuál de las tres pertenece ni cuando empieza o acaba la lista de cada una de ellas. Mi solución se basa en dos estrategias, retrasar la decisión del analizador y hacer factor común.

Estas estrategias las lleve a cabo con las construcciones que, aunque tenían distinciones (como el token constant en las declaraciones de constantes), el analizador no sabía a cual pertenecía el identificador. Así que, como se ve en las reglas listaDeclaraciones y lista1, retraso la elección del analizador leyendo los tokens problemáticos mientras junto las declaraciones para que solo tenga un camino valido el analizador. Después de comprobar a quien pertenece el identificador, pasamos a la siguiente lista en la que, teniendo en cuenta el orden en el que se tienen que producir las declaraciones, la lista de declaraciones es más pequeña y fácil de manejar para el analizador.  Comentar, por ejemplo, el camino seguido para la declaración de una constante. Al analizador le llega un token IDENTIFICADOR, lo capta con la regla listaDeclaraciones y se va ha lista1. En lista1 vemos que lee el token DOSPUNTOS (el cual siguen compartiendo varias declaraciones) y se va a lista2. Es aquí en lista2 donde el analizador tiene que tomar una decisión, y, en este ejemplo, leería el token CONSTAN por lo que lee declaración del constante al completo para acabar en una lista de constantes, posiblemente vacia, y declarar todas las constantes que haya. Cuando acabamos con las constantes nos vamos a otra lista (listaDeclaracionesTipo), en los que ya no es legal declarar constantes y solo se pueden declarar de los otros tipos de declaraciones.

Antes de seguir me gustaría entrar en el caso de la declaración de tipos primitivos y mi decisión en la implementación. Aun usando las técnicas de antes, entre la declaración de tipos primitivos y la de variables sigue habiendo ambigüedad debido a que empiezan con IDENTIFICADOR y DOSPUNTOS. Al final tuve que optar para conseguir declarar todos los tipos primitivos que entrara en un buble entre listaDeclaracionesTipo, lista3 y lista4 para que se fuera leyendo estas declaraciones, hasta que no hubiera mas o entrara otro tipo de declaración válida.

Con las diferentes estrategias propuestas anteriormente sigo en esta sección de reglas de producción hasta que no queda ambigüedad y continuamos con el resto de las reglas.

El resto de las reglas seguí sin problemas y no hay nada demasiado destacable que comentar, hasta que llegamos a las expresiones. Con las expresiones me paso lo mismo que con las declaraciones, al intentar "juntar" las diferentes piezas empezaros a saltar ambigüedades y errores. En este caso tuve que tomar un par de decisiones por el bien y la simplicidad de las reglas.

La primera es que, al estar en el análisis sintáctico, es muy difícil saber cuando una expresión que consiste simplemente en un identificador (por ejemplo "x") es una expresión aritmética o una expresión lógica. Al no poder quitar esta ambigüedad, decidí crear un tipo diferente de expresiones, llamada "exprGeneral", el cual solo está compuesto por el token IDENTIFICADOR. Con esta nueva expresión me puedo centrar en las demás expresiones tanto aritméticas como lógicas y dejarle a un futuro analizador semántico que identifico de que tipo son estas expresiones generales.

No contento con esto, las expresiones lógicas contienes también expresiones aritméticas con operadores lógicos, lo cual llevo a mas ambigüedad. La solución es parecida a la de las declaraciones, retrasar la decisión del analizador para que con cada token pueda elegir solo un camino correcto.

Además de estos dos bloques con errores y ambigüedades, el resto de la parte del analizador sintáctico lo realicé sin más problemas, fijándome bien en las diferentes reglas que debía implementar.