



TRADUCTOR DE PROGRAMAS: PASCAL a C

**Introducción**

El objetivo de la práctica es transformar un lenguaje parecido a PASCAL en un lenguaje parecido a C usando un traductor. Para lograrlo se ha divido el trabajo en dos partes, un analizador léxico (jflex) y uno sintáctico (cup) unidos por un programa java. Para compilarlos usamos el cmd de Windows.

**Analizador Léxico**

La función del AL es, a partir de un conjunto de reglas léxicas, reconocer las cadenas que cumplen dichas reglas y enviar la información al analizador sintáctico.

La especificación léxica se divide en 3 partes, separadas por ‘%%’:

1. Código de usuario

Código java auxiliar necesario para que funcione el cup.

1. Opciones y declaraciones

Contiene opciones de código generado, como el nombre que queremos que tenga la clase resultante, que nos muestre la línea y la columna, opciones de compatibilidad con cup y que sea caseless, ya que Pascal es un lenguaje no-case sensitive y obvia si los caracteres están en mayúscula o minúscula, debe aceptar ambos.

También contiene la declaración de un estado exclusivo llamado STRING, los macros y código java auxiliar. Hay macros para espacios y saltos de línea, dígitos, identificadores y cadenas de carateres entre otros. En la zona de código se declara la clase Symbol, necesaria en las acciones.

1. Reglas léxicas

Se definen los patrones asociados a los estados que debe reconocer el analizador léxico.

Vemos que la mayoría de reglas que aparecen no están asociadas explícitamente a ningún estado, se asocian automáticamente al estado por defecto, un estado inclusivo llamado <YYINITIAL>. Primero, definimos las reglas para los símbolos terminales sencillos y las acciones que se van a tomar, que son devolver una instancia de la clase Symbol asociado a cada símbolo, y después, las reglas para las cadenas de caracteres, identificadores y constantes numéricas. Los espacios y comentarios se ignoran.

A continuación, definimos las reglas del estado STRING. Si se encuentra el token `, se pasa a este estado. Una vez dentro, el analizador no busca los patrones anteriores, sino solo los específicos de esta clase y controla que haya otro **‘** para cerrar la constante literal y volver al estado anterior. Si se detectan dos **‘‘** seguidos se transforman en uno solo como indica el enunciado.

Por último, se define la recuperación de errores léxicos. Si no se reconoce ninguna de las reglas anteriores, se notifica el error indicamdndo el carácter no reconocido y su ubicación.

**Analizador Sintáctico**

La función del AS es añadir acciones semánticas al traductor. El nuestro sigue el esquema de definición dirigida por sintaxis: las acciones se asignan a consecuentes completos.

Primero redefinimos las funciones para la recuperación de errores en la sección de *parser code.* A continuación, declaramos los símbolos terminales y los no terminales. La gramática tiene reglas ambiguas, por lo que tenemos que definir la precedencia y asociatividad de los operadores. Para finalizar esta sección, indicamos al analizador que debe empezar por la regla **PRG**.

A continuación definimos las reglas semánticas y las acciones asociadas. En un resumen rápido, hemos definido las acciones semánticas de manera que, empieza desde los no terminales más básicos, como **OPCOMP** o **OPARTIT** y va sus valores hacia arriba en el árbol, acumulando los valores hasta llegar a **PRG**.

Se han usado tres clases para guardar los valores, los más básicos son los Símbolos, estos se guardan en Sentencias, las cuales se guardan en dos listas en la clase Función, cada Sentencia en las listas representa una línea. Las dos listas se llaman Declaraciones y Bloque, pensado así para que todas las funciones y métodos se guarden en Declaraciones según va subiendo el traductor. Esto nos permite saber que parte del programa ira dentro del *main*, ya que ira dentro de la lista Bloque.

Una vez llegamos a **PRG** en el traductor, utilizamos el método reestructurar de la clase Función para imprimir las sentencias traducidas como deseamos. Este método recorre ambas listas, imprimiendo cada símbolo. Al principio de cada sentencia añade un numero de tabulaciones, el cual está determinado por si la sentencia actual es un cierre (baja el numero) o una apertura (lo sube).

Para saber cuando debemos transformas una asignación en un *return*, metemos en un *String* el nombre de la función actual y si alguna asignación dentro de él repite ese nombre lo cambiamos por *return* y ponemos un *boolean* a false para avisar al programa que durante lo que queda de sentencia no debe imprimir nada salvo lo que se está retornando.

Un problema encontrado es que el lenguaje no funciona como debería, al traducir no distingue entre **PROC\_CALL** y **FACTOR**, lo cual no seria un problema si no fuera porque cuando se llama a un método sin ningún valor no detecta el **SUBPARAMLIST** y no devuelve los paréntesis para indicar que una llamada.

Para solucionar esto se ha optado por meter una lista que se va llenando con los nombres de todos los métodos a los que no se les pasa ningún valor y si se vuelve a detectar esa ID se pone el “()”. Esto se hace si falla el *if* que comprueba si debe ponerse un *return* para así evitar confusiones.

**Ejemplos**

**Me queda explicarlo**

**Captura de pantalla de 2019-06-23 16-09-17**

**Captura de pantalla de 2019-06-23 16-09-40**

**Captura de pantalla de 2019-06-23 16-10-35**

**Captura de pantalla de 2019-06-23 16-11-34**

**Bibliografía**

<http://www2.cs.tum.edu/projects/cup/docs.php>

<https://jflex.de/manual.html#ExampleUserCode>

<https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>

<http://pages.cs.wisc.edu/~fischer/cs536.s06/course.hold/html/NOTES/4a.JAVA-CUP.html>