Universidad
Rey Juan Carlos

PROCESADORES DE LENGUAJES

Grado en Ingeniería Informática

Práctica obligatoria

(convocatoria extraordinaria)

Ismael Vicente Rodríguez

Alberto Pérez Pérez

Rodrigo Martínez Ruiz

Repositorio en GitHub:
https://github.com/cfkr-dev/ANTLR-Code-Summary

Móstoles (Madrid) a Julio de 2023

Índice

1. Descripción del Trabajo Realizado

1.1. Parte Obligatoria

1.1.1. Breve explicación

El código del proyecto comienza con la especificación de la gramática que se encargará del análisis sintáctico, en la que se transcribe la gramática aportada por los profesores a código siguiendo una notación BNF y transformándolo en la medida de lo posible a una gramática de tipo LL(1).

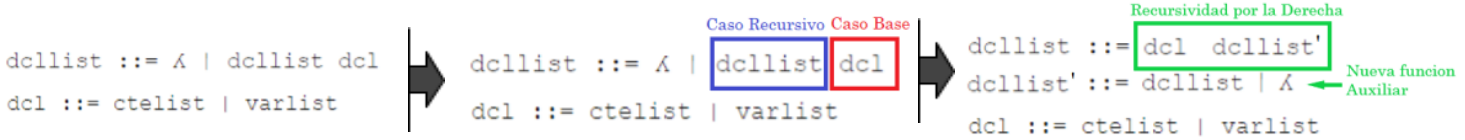
En la segunda parte del código se encuentra la especificación de tokens donde "ANTLR" tratará el análisis léxico. Sobre las reglas de la especificación sintáctica se ha procedido a hacer una traducción dirigida por sintaxis que se apoya en una estructura de clases y sus herencias para simplificar y modularizar el proyecto.

1.1.2. Proceso de conversión a LL(1)

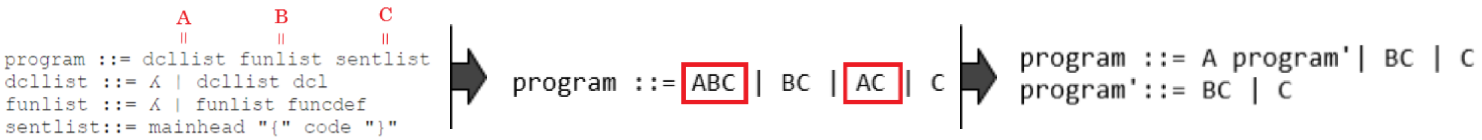
El proceso de conversión que se ha seguido ha sido primero localizar qué puntos de la gramática no cumplen con el formato de tipo LL(1), ya que en nuestra gramática originalmente, se daban casos de recursividad por la izquierda y se daban casos en los que los directores no podrían diferenciarse por un solo símbolo director.

Para ello se ha seguido en toda la gramática el siguiente sistema que veremos en un breve ejemplo para tratar de resolver estos problemas:

- Caso Recursivo por la Izquierda básico



- Caso Factor Común



Se ha realizado este proceso en toda la gramática según se han ido encontrado casos que tuvieran estos problemas impidiendo que fueran del tipo LL(1)

Si se hace un análisis en profundidad, calculando los conjuntos directores nos damos cuenta que tras muchas derivaciones esta gramática tiene algunos directores que no pueden diferenciarse a través de un símbolo director. Estos casos no se ven a simple vista y se dan en ciertas derivaciones, de manera indirecta por lo que no se han tenido en cuenta estas.

1.1.3. Traducción dirigida por Sintaxis

El objetivo de la traducción dirigida por sintaxis en este caso era transformar el código que interpretaba la gramática a formato "HTML" mediante acciones semánticas.

Para evitar sobrecargar la gramática con código encargado de transformar en "HTML" e imprimirlo, las acciones semánticas se centraron en llamar a métodos que se encargaban de almacenar todos los elementos leídos por la gramática y posteriormente imprimirlos utilizando métodos propios de "HTML" según su tipo.

Estos métodos pertenecen a clases almacenadas en la carpeta "Semantic".

Es importante destacar que el sistema de clases implementado es bastante denso, ya que se requiere una clase para cada posible caso de código que la gramática pueda generar. Es fundamental tener en cuenta el contexto de estas clases para realizar la conversión a "HTML", ya que el contexto proporciona la mayor parte de la información necesaria. Por ejemplo, el contexto nos indica en qué parte del programa se ha creado una función o una variable. Siguiendo esta idea, se han creado distintas interfaces que generan las clases según si el contexto es el programa principal, una función auxiliar , la función main o una estructura programable local. De esta manera, cada vez que se crea una función, se crea un nuevo contexto en el que el código perteneciente a ese contexto sabrá, en el caso de las variables locales, que pertenecen a esa función, y en el caso de usar variables globales, que se han declarado en otro contexto.

Dentro de las clases mencionadas, se ha decidido diferenciar entre elementos del programa y elementos programables, que permiten la programación dentro de sí mismos mediante la apertura de llaves ("for", "while", "if"...). Esta distinción permite diferenciar entre una variable local y una variable del programa.
Entrando más en detalle de lo mencionado para orientarse dentro de todas las clases que se han creado y utilizado, lo primero es localizar donde están

estas clases, que se encuentran dentro de la carpeta "semantic" en distintas subcarpetas según el tipo de elemento que sean. Nosotros destacaremos las siguientes clases que son el núcleo del funcionamiento de las clases creadas que están conectadas entre ellas a través de herencia, pero cuyo funcionamiento es la parte principal empleada en las acciones semánticas realizadas en la gramática. Estas clases son:

- "MasterSentenceContainer": Esta interfaz representa el contexto del tipo sentencia que se encarga de crear las clases que representan todo tipo de sentencias, entre ellas , cabe destacar las que usan expresiones que tratan a su vez con las interfaces "*MasterBinaryOperation*" y "*MasterUnaryOperation*".
- "Program": Se trata de una clase que representa el programa principal y es el primer contexto que se crea, a partir del cual se generan el resto
- "Function": Esta clase es la encargada de representar y generar los elementos de tipo función, y crear un nuevo contexto a través del cual el contenido de esta, reconozca que pertenece a dicha función.

Para una mayor aclaración de cómo se relacionan algunas de estas clases importantes, se ha realizado un "UML" que relacionan las interfaces con las clases abstractas principales con las que se han trabajado.No obstante recomendamos quedarse con la idea mencionada anteriormente.

("UML" en Anexo)

Todas estas clases tienen un método llamado "toHTML()". Este método es el que nos permitirá generar un String con el resumen del programa que posteriormente meteremos en un fichero. Dado que almacenamos toda la información de los tokens en objetos, estos objetos se estructuran en forma de árbol, cada objeto sería un nodo con una referencia al padre y una referencia a cada elemento que contiene. La raíz de este árbol sería un objeto de la clase "Program", una vez se haya terminado de analizar la gramática y generar el árbol, se procede a invocar el método "toHTML()" de "Program", este genera un string e incluye lo necesario para formar la estructura básica del fichero ".html", también invoca el método "toHTML()" de los hijos que devolverán su parte del fichero. Generando de forma recursiva el fichero.

Cabe destacar que la creación de los elementos HTML se ayuda de una clase *"helper"* ("semantic.utils.HTMLHelper.java") la cual está integrada por un repertorio de funciones de creación de "strings" siguiendo la estructura de las etiquetas HTML requeridas.

1.2. Parte Opcional

De la parte opcional hasta el momento se ha realizado la incorporación de las sentencias de control, es decir la ampliación de la parte sintáctica. Se han incorporado todas ellas y en algunas se han modificado algún cambio extra para acercarlas más al lenguaje que trata de representar. Estos últimos cambios se pueden ver en el apartado 1.3.

Sobre la corrección de errores no ha hecho falta modificar la gramática, puesto que "ANTLR" ya tiene incorporado un sistema de corrección de errores, por lo que al percatarnos y entender su funcionamiento, no ha sido necesario que realizaramos nada nuevo.

Se ha añadido en la carpeta error una clase que se encargue de imprimir los mensajes de error que detecte la gramática, indicando fila, columna y posible causa. La implementación seguida no es más que una generada en base a la que nos proporciona la guía "ANTLR" (The Definitive ANTLR 4 Reference). Por otro lado se han modificado los mensajes de errores que lanza la clase "DefaultErrorStrategy" de "ANTLR" en caso de detectar algún tipo de error sintáctico, de modo que ahora todos se muestran en español. En la siguiente imágen puede apreciarse esta característica.

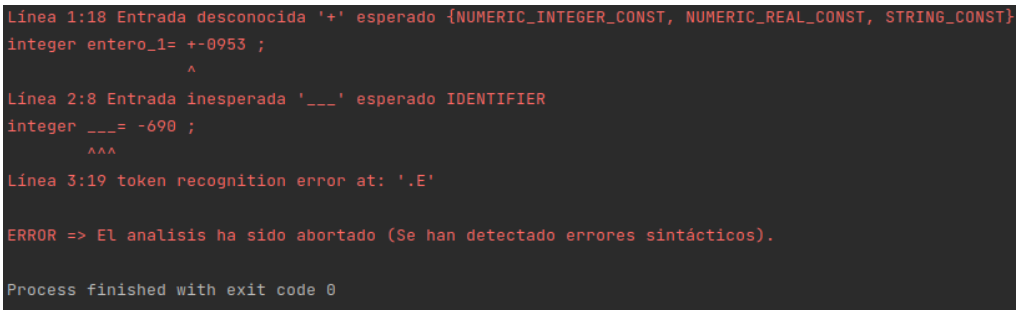


Imagen 1 . Captura de salida de error por consola con errores sintácticos

Para la parte opcional del apartado de traducción dirigida por sintaxis, la implementación se ha llevado a cabo conjuntamente a la parte obligatoria, de forma que el funcionamiento de esta es igual a la mencionado en la parte obligatoria en el apartado Traducción dirigida por Sintaxis.

Respeto a la implementación que sigue la traducción dirigida por sintaxis, se ha añadido un sistema de reconocimiento de errores semánticos en el que el programa es capaz de detectar qué tipos de datos son compatibles con otros para por ejemplo operar con ellos, o que tipo de expresiones son por ejemplo, consideradas válidas como condiciones para una sentencia "if".

Cabe destacar, a modo de información, que todos los aspectos relacionados con el tipado de los elementos del lenguaje se gestiona mediante información guardada en mapas dentro del fichero de constantes ("semantic.utils.Constants.java"), de modo que si una regla de

tipado desea ser cambiada, solamente hay que quitar o añadir elementos a estas estructuras, manteniendo así un diseño escalable.

En la siguiente imágen se muestra el correspondiente funcionamiento, el cual también puede ser apreciado en la ejecución del código de ejemplo número 8.

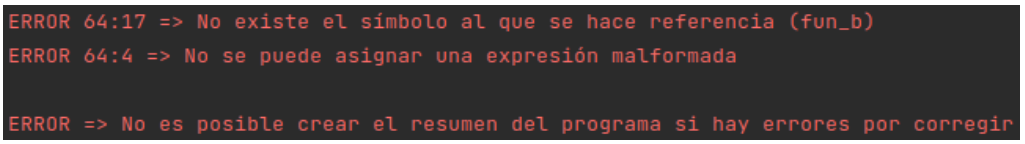


Imagen 2 . Captura de salida de error por consola con errores semánticos

1.3. Cambios a tener en cuenta

1.3.1. Modificación de la sentencia return

Se ha modificado el punto de retorno de las funciones para que su uso sea más parecido al de un lenguaje real.

1.3.2. Cambio en los argumentos en la llamada a funciones

Ahora cuando se llama a una función puede tener argumento vacío, además que los argumentos de la función también permiten expresiones como permiten otros lenguajes.

1.3.3. Modificación de los structs

Se han modificado los "`structs`" de tal manera para que puedan ser llamados desde una función, y se ha mejorado como la gramática trabaja con ellos.

1.3.4. Modificación en el bucle for para hacerlo más completo

Se ha modificado la gramática del bucle "`for`" para que se pueda declarar la variable índice dentro del bucle como en el lenguaje real.

1.3.5. Declaraciones en línea

Se modificó la gramática para que se puedan hacer que se puedan declarar y asignar en una misma línea variables.

1.3.6. Sistema de casting

Gracias a la gestión del tipado con el módulo "`Semantic`" podemos realizar una acción de *"casting"* sobre elementos del lenguaje en caso de que dicho elemento forme parte de un argumento de una función con el nombre de un tipo. Por ejemplo, supongamos una variable de tipo "`string`", podemos realizar, "`FLOAT(variable)`", y automáticamente el programa la detecta como tipo "`FLOAT`" (siempre y cuando el casting fuese compatible).

2. Casos de Prueba

2.1. Entradas correctas

2.1.1. Código 1

En este código se muestra un código básico general para ver un correcto funcionamiento global, en el que hay comentarios de ambos tipos, hay declaraciones de funciones y sus argumentos, operaciones y sentencias de control de flujo anidadas.

2.1.2. Código 2

Esté código es un breve ejemplo de cómo se usan los `"structs"`, incluyendo un anidamiento de estos. Además se comprueba alguna otra sentencia de control

2.1.3. Código 3

El código de ejemplo 3 se basa en mostrar primero el funcionamiento correcto del léxico, en identificadores y los valores de las constantes numéricas. Además vemos que no afectan los saltos de línea o espacios entre sentencias y los sigue considerando una sola.

2.1.4. Código 4

En este código de ejemplo 4 muestra el funcionamiento correcto de los `"STRING_CONST"` y sus casos especiales en profundidad. Además se muestra también en profundidad, el funcionamiento tanto de los comentarios monolinea como de los comentarios multilínea y diferentes casos de uso como el uso interesantes.

2.2. Entradas Incorrectas

Los códigos incorrectos puestos como ejemplo no son más que una copia de los correctos, pero mostrando en los mismos puntos a destacar de los correctos los casos de error y cómo se detectan.

2.2.1. Código 5

En este ejemplo nos centramos en quitar elementos como cierre de paréntesis o corchetes(en el primer `"else"`), sentencias de control de flujo mal compuestas (el último `"if"`) o comentarios mal creados.

2.2.2. Código 6

Este código solo trata de representar algún error en la declaración de la sentencia de control, se han eliminado puntos y comas y se han producido algunos errores de escritura.

2.2.3. Código 7

Este código muestra la detección de errores de tipo léxico, que podemos ver al principio en la declaración de variables, tanto en el tipo de identificador como en los valores que pueden tomar las constantes. Estos errores se producen por no seguir el formato que impone el léxico. Además se muestra casi al final en la línea 31 un caso en el que el léxico no es capaz de diferenciar si se trata de un número con el símbolo delante o una expresión aritmética a falta de un espacio o un símbolo extra que confirme el formato como veíamos en el código de ejemplo 3 en la misma operación.

2.2.4. Código 8

Este código ejemplifica los posibles errores que el módulo semántico puede detectar, mostrando estos por consola. (Para visualizar dichos errores, leer el comentario al inicio del fichero).

3. Anexo

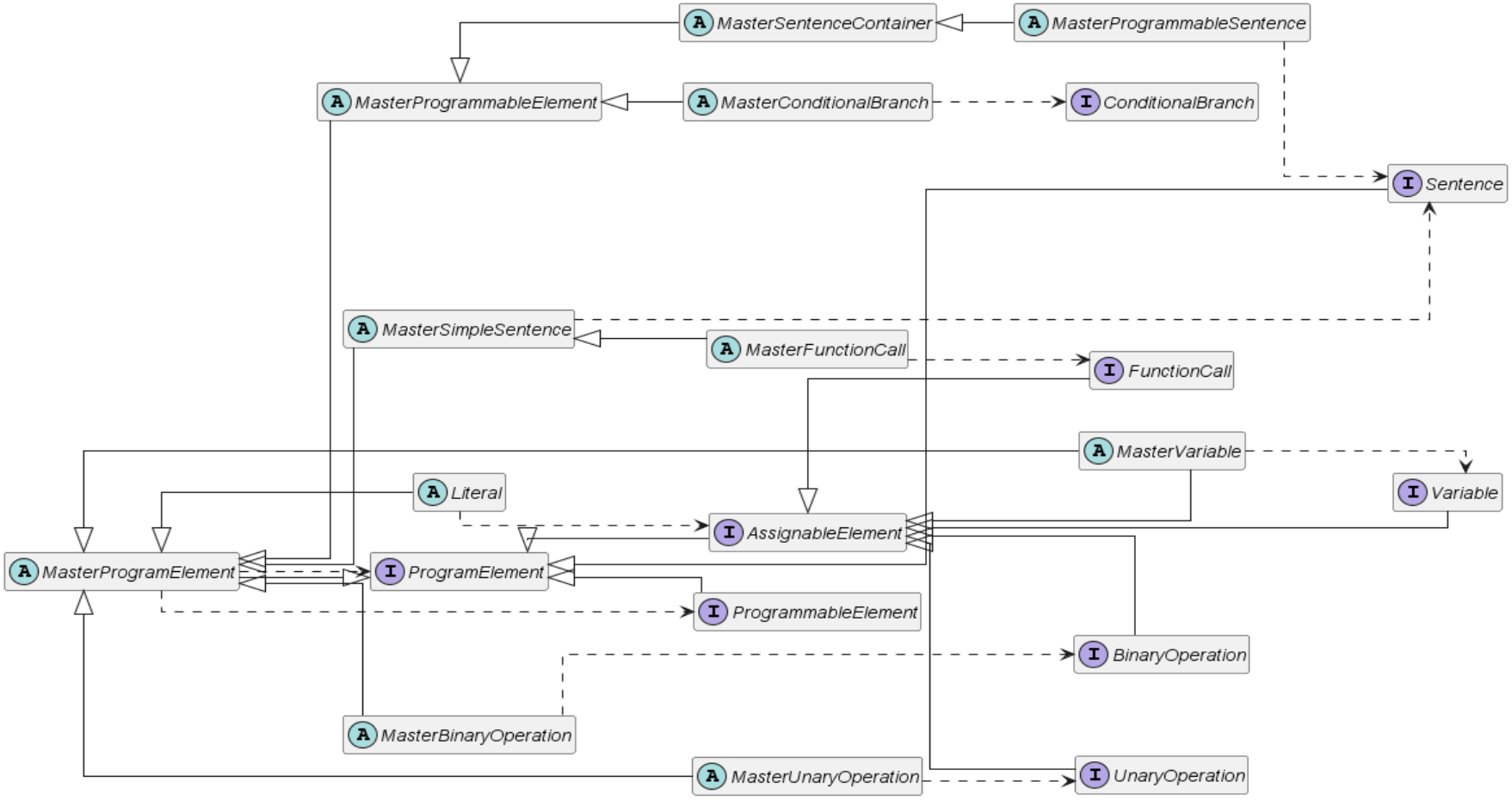


Imagen 3 . "UML" reducido de las interfaces y clases abstractas representativas