Práctica. Segunda Fase

Desarrollo de analizadores sintácticos para Tiny(0) y para Tiny

En esta segunda parte debe realizarse el siguiente trabajo:

1) Desarrollo manual de un analizador sintáctico para **Tiny(0)**. Para ello, deberá entregarse:
   - Un apartado en la memoria con las siguientes secciones:
     - Especificación sintáctica (gramática) para **Tiny(0)**. Dicha especificación deberá desarrollarse utilizando los patrones de escritura de gramáticas explicados en clase.
     - Acondicionamiento de la gramática para permitir la implementación de un analizador sintáctico descendente predictivo recursivo.
     - Directores de cada regla de la gramática acondicionada.
   - Una implementación manual, en Java, del analizador sintáctico descendente predictivo recursivo. Dicho analizador deberá funcionar con la implementación manual del analizador léxico para **Tiny(0)** desarrollado en la primer fase. Debe proporcionarse, además, un programa de prueba que acepte como argumento el archivo a procesar, y genere como salida, bien un mensaje legible del primer error (léxico o sintáctico) detectado, bien un mensaje "OK" cuando el programa analizado sea sintácticamente correcto. Dicha implementación deberá estar, además, acondicionada para su funcionamiento y prueba a través del juez (**apéndice A)**.
2) Desarrollo de analizadores sintácticos descendentes y ascendentes para **Tiny**. Para ello, deberá entregarse:
   - Un apartado en la memoria con las siguientes secciones:
     - Especificación sintáctica (gramática) para **Tiny**. Dicha especificación deberá desarrollarse utilizando los patrones de escritura de gramáticas explicados en clase.
     - Acondicionamiento de la gramática para permitir la implementación de un analizador sintáctico descendente predictivo recursivo.
   - Una implementación de un analizador sintáctico descendente predictivo desarrollada con **javacc**. Dicha implementación deberá venir acompañada de un programa principal que producirá como salida, bien un mensaje legible del primer error (léxico o sintáctico) detectado, bien un mensaje "OK" cuando el programa analizado sea sintácticamente correcto.
   - Una implementación de un analizador sintáctico ascendente desarrollada con CUP. Dicha implementación integrará el analizador léxico para **Tiny** desarrollado con **jflex** en la primera fase de la práctica. Dicha implementación deberá venir acompañado de un programa principal análogo al pedido para el analizador desarrollado con **javacc**.

Ambas implementaciones (tanto la basada en **javacc** como la basada en **cup**) deberán estar acondicionadas para su funcionamiento y prueba a través del juez (**apéndice A**).

En la memoria que deberá entregarse con esta práctica habrá que incluir una portada en la que aparezcan los nombres y apellidos de los integrantes del grupo, y el número de grupo.

Fecha límite de entrega: **Viernes 7 de marzo de 2024, a las 23:59h**.

Modo de entrega: A través del campus virtual, en un único .zip. Dicho archivo debe contener: (i) un documento PDF `memoria.pdf` con la memoria; (ii) una carpeta `implementación_tiny0`, en el interior de la cuál debe incluirse toda la implementación requerida para **Tiny(0)**; (iii) una carpeta `implementación_tiny_javacc`, en el interior de la cuál debe incluirse la implementación **javacc** requerida para **Tiny**; (iv) una carpeta `implementación_tiny_cup`, en el interior de la cuál debe incluirse la implementación **cup** requerida para **Tiny**; (v) una carpeta `pruebas_tiny_0` con distintos programas de prueba que permitan probar la implementación para **Tiny(0)**; y (vi) una carpeta `pruebas_tiny` con distintos programas de prueba que permitan probar la implementación para **Tiny**.

La entrega debe ser realizada solamente por un miembro del grupo.

Las implementaciones deberán, además, entregarse a través del juez DomJudge de la asignatura.

Apéndice A Acondicionamiento DomJudge de los analizadores sintácticos

1) **Salida esperada por el juez**

Deben escribirse los lexemas de la secuencia de tokens procesada, con el mismo formato utilizado en la fase 1. En caso de que se detecte un error léxico, debe escribirse `ERROR_LEXICO` (y la salida termina). Si se detecta un error sintáctico, debe escribirse `ERROR_SINTACTICO` (y la salida termina). Por tanto,

teniendo en cuenta que nuestros analizadores nunca *desplazan* un token erróneo (es decir, siempre consumen prefijos de sentencias generadas por la gramática):

- En caso de que no haya errores, se imprimirá toda la secuencia de tokens de la entrada (como se hacía en la prueba del analizador léxico).
- En caso de que haya errores, se imprimirá los tokens que preceden a la posición en la que se ha detectado error, y `ERROR_LEXICO` o `ERROR_SINTACTICO`, dependiendo del tipo de error detectado (y no se imprime nada más, la ejecución se interrumpe aquí).

Importante: las palabras reservadas se escribirán utilizando el mismo formato que en la fase 1: `<int>`, `<true>`, … El fin de fichero se escribirá como `<EOF>`, lo mismo que en la fase 1.

2) Preparación y empaquetado del analizador manual para Tiny(0)

Lo primero que hay que hacer es integrar el analizador sintáctico manual, con el analizador léxico, con el gestor de errores. Para tal fin, el gestor de errores puede pasarse en el constructor del analizador léxico (esto se puede hacer en el constructor del analizador sintáctico). De esta forma, los errores léxicos se notifican invocando al método `errorLexico` del gestor de errores, mientras que los errores sintácticos se notifican invocando al método `errorSintactico`.
Lo segundo es que el gestor de errores, en lugar de imprimir mensajes y terminar, debe levantar excepciones: `ErrorLexico` en el método `errorLexico`, `ErrorSintactico` en el método `errorSintactico`.
Lo tercero es equipar al analizador sintáctico con la capacidad de trazar los emparejamientos. Para ello, cuando se empareja con éxito un token, desde el método `empareja` se invoca a un nuevo método protegido llamado `traza_emparejamiento`, al que se pasa el token emparejado (es decir, `anticipo`). Con ello, es posible especializar el analizador sintáctico (v.g., mediante una clase `AnalizadorXXX`**`DJ`**, que hereda de la clase `AnalizadorXXX` que implementa el analizador sintáctico) redefiniendo el método `traza_emparejamiento` para que imprima de manera apropiada el token emparejado. (en `AnalizadorXXX` dicho método no hará nada).
Por último, debe proporcionarse una clase principal, `DomJudge`, en el paquete por defecto, que defina un método `main` que lance el analizador sintáctico, usando `System.in` como entrada.
Todas las fuentes del proyecto deben empaquetarse en un .zip, que será el que se envíe al juez (al igual que ocurría con el analizador léxico en la fase 1), eligiendo `procleng` como lenguaje.
En el Campus Virtual, en el código de ejemplo, podéis encontrar una implementación del analizador sintáctico para Eval, que usa la implementación manual del analizador léxico, y que ha sido preparado como se ha descrito, para que pueda ser ejecutada en el juez.

3) **Analizador javacc**.

El método que vamos a seguir es similar al usado en el analizador manual. Para ello, debemos definir en el archivo de especificación .jj la opción **DEBUG_PARSER=true;** Con ello, cada vez que el analizador empareja un token, invoca al método protegido `trace_token`, pasándole como primer argumento el token emparejado (el segundo argumento no es relevante en este contexto).
Esto nos permite definir una subclase que especializa a la clase generada, y en la que vamos a sobrecargar el método `trace_token`. Por ejemplo, si la clase que implementa el analizador sintáctico es `AnalizadorXXX`, nuestra clase será `AnalizadorXXX`**`DJ`**. Allí redefiniremos el método `trace_token` para imprimir el token de manera apropiada. Hay que tener en cuenta que, en el caso de que el token sea una palabra reservada, tenemos que imprimir su forma canónica. El campo `kind` del token nos dice cuál es su clase léxica. Además, como `AnalizadorXXX` implementa la interfaz con las constantes donde se definen las clases léxicas, tendremos disponibles todas esas constantes. Un *switch* hace el resto, con una opción para cada palabra reservada y un *print* adecuado, una opción para EOF y un *print* apropiado, y, para el resto, un *print* del lexema (el lexema está en el campo `image` del token).
En el constructor de `AnalizadorXXX`**`DJ`** es necesario también deshabilitar el resto de la traza (no hay que olvidar que hemos generado el analizador sintáctico en modo 'depuración'). Esto se hace invocando al método `disable_tracing`.
Una vez hecho esto, podemos proporcionar una clase `DomJudge` análoga a la del caso manual, que instancie el analizador sintáctico, pasándole `System.in` como su entrada, e invoque el método de inicio. En este caso, la excepción que anuncia un error léxico es `TokenMgrError`, y la excepción que anuncia un error sintáctico es `ParseException`.

Por último, es necesario indicar al juez la ruta (relativa) de la carpeta en la que está la especificación. Esta ruta se escribe en un archivo `ruta_javacc`. La especificación *javacc* en sí tiene que estar en un archivo llamado `spec.jj`.
Todo esto se empaqueta en un .zip, que es el que hay que subir al juez, eligiendo `procleng` como lenguaje.
En el Campus Virtual, en el código de ejemplo, se puede encontrar una implementación *javacc* del analizador sintáctico para `Eval`, preparada y empaquetada como he descrito, para que pueda ser ejecutada en el juez.

4) **Analizador cup.**

El método es similar al seguido en **javacc**. En este caso, la clase `lr_parser`, del *runtime* de **cup**, de la que heredan todos los analizadores generados por cup, define un método '**debug_shift**' que es invocado con cada desplazamiento cuando el analizador se ejecuta en modo de depuración (con el método `debug_parse`). Con este método podemos imprimir los tokens desplazados por el analizador. Para ello, creamos una subclase `AnalizadorXXX`**`DJ`**, que hereda de la clase `AnalizadorXXX` generada por **cup**, y redefinimos el método `debug_shift` (que recibe el `Symbol` emparejado) para que escriba el lexema del correspondiente símbolo (campo `.value`). Así mismo, para deshabilitar la escritura de otros mensajes, tenemos que redefinir el método `debug_message` (también definido en `lr_parser`, e invocado en modo *depuración*) para que no haga nada.
La clase `DomJudge`, en este caso, deberá invocar al analizador sintáctico mediante el método `debug_parse`.
En cuanto a la propagación de los errores, se sigue la misma estrategia que en la implementación manual: los métodos del gestor de errores levantan las correspondientes excepciones, que se capturan en el `main`, imprimiendo `ERROR_LEXICO` o `ERROR_SINTACTICO` según proceda (y terminando la ejecución).
Para conseguir la salida en el formato esperado, en ALexOperations hay que indicar el lexema normalizado para las palabras reservadas y para EOF, en la construcción de los respectivos *tokens*.
Por último, aparte de `ruta_jflex`, hay que poner, en la raíz del paquete, un par de archivos adicionales:

- `ruta_cup`, con la ruta relativa de la carpeta donde está la especificación para **cup**.
- `cup_options`, con las opciones de generación de CUP relativas al nombre del analizador y del contenedor de las clases léxicas. Por ejemplo:

  ```
  -parser AnalizadorSintacticoXXXX -symbols ClaseLexica
  ```

  o cualesquiera otros nombres que se quiera utilizar.

De nuevo todo esto puede empaquetarse y enviarse al juez (lenguaje `procleng`).
En el Campus Virutal hay un ejemplo completo de implementación CUP para el analizador de Eval, preparado y empaquetado como hemos descrito.

5) **Problemas de entrega.**

- Las implementaciones manual, *javacc* y *cup* de ejemplo para el analizador sintáctico para EVAL pueden probarse todas ellas en el problema `EJEMPLO-PARSER`.
- La implementación del analizador para Tiny(0) deberá enviarse a `03 ANALIZADOR-SINTACTICO-TINY(0)`
- La implementación del analizador *javacc* para Tiny deberá enviarse a `04 ANALIZADOR-TINY-JAVACC`
- La implementación del analizador *cup* para Tiny deberá enviarse a `05 ANALIZADOR-TINY-CUP`