Universidad
Carlos III de Madrid

Práctica 1

Familiarización con el análisis léxico y sintáctico (Parte 2 - Análisis Sintáctico)

Procesadores del Lenguaje

Carlos Dumont Cabrilla – 100074907

Universidad Carlos III de Madrid

Table of Contents

1 Introducción

Se ha de desarrollar un reconocedor y procesador de un lenguaje regular con una gramática acorde que permita reconocer distintas sentencias, desde una simple calculadora de enteros, pasando por una de reales y un reconocedor de comentarios y palabras clave, hasta reconocimiento de declaración de variables y almacenamiento de estas en memoria.

Archivos clave:

- **lexer.jflex:** definición del análisis léxico
- **parser.cup:** definición del análisis sintáctico
- **input.txt:** declaración de pruebas con las palabras a reconocer

JDK: Java SE 11

2 Parte 1. Análisis Léxico

En esta sección se aborda el análisis léxico del lenguaje, donde se declararan los patrones **Regex** con los que reconocer como tokens las palabras a reconocer del archivo input.txt.

Para poder comunicar el **lexer** con el parser, el lexer "exporta" los tokens reconocidos o sus alias incluyendo en los casos que sea necesario, una variable con el token reconocido y en el adecuadamente tipado. Estos **tokens y alias (símbolos)** "exportados" deben coincidir con los declarados en la **tabla de símbolos** (que a su vez coincidirá con los del parser sintáctico).

Basado en algunos de los ejemplos del enunciado, todas las sentencias se diseñado para esperar un cierre con ";", excepto en el Ejercicio C, donde las palabras clave a reconocer no esperan un cierre con ";" por no estar indicado.

Ejercicio A

Objetivo

Aceptar e ignorar **comentarios**, que deben ser tratados –e ignorados– por completo en el análisis léxico.

Los comentarios comenzarán con los caracteres ' <!-- ' y se extienden hasta la aparición de '--> ', pudiéndose extender a varias líneas separadas.

Solución

Se reconocen los literales de apertura y cierre del comentario y en su interior la posibilidad repetida de 0 o n caracteres cualesquiera dentro, incluyendo los saltos de línea. En conjunto, el comentario es reconocido e ignorado.

Tokens

- *ModernComment = "<!--" (. | {Newline})* "-→"*

Tests

- *27+15;*
- *<!-- Hello*
- *3+2;*
- *World -->*

Ejercicio B

Objetivo

Incorporar **números reales** (con notación científica habitual) y números **hexadecimales** (comienzan por la secuencia “ 0x ” o “ 0X ”).

Solución

Se reconocen tanto números decimales tanto en notación común como notación científica (incluyendo posibilidad de “e”, “E” y signo “–“ ó “+” delante). Se aprovecha que su concepto, significado sintáctico y tipo (Double) es el mismo, para generar un alias común, que los unifica de cara al parser, llamado DoubleNumber.

También se reconocen números en formato hexadecimal (base 16) tanto con formato de inicio “0x” como “0X”. Para ser parseado correctamente con Java, es necesario reconocer “0x”/”0X” pero extraerlo de la cadena de texto para parsear únicamente el número sin la cabecera.

Además, se han añadido (respecto al modelo de ejemplo de la librería CUP): la operación “/” para dividir, dando 0, n, n.n ó Infinity según el caso adecuado, y reservado la negación de los números exclusivamente a “-”.

Tokens

- *RealNumber = {Number} "." {Number}*
- *ScienceNumber = ({Number} | {RealNumber}) ("e" | "E")("-" | "+" | "") {Number}*

- *DoubleNumber = {RealNumber} | {ScienceNumber}*
- *HexAlfaNumber = ([0-9A-F])+*
- *HexNumber = "0x" {HexAlfaNumber} | "0X" {HexAlfaNumber}*

Tests

- *-2;*
- *3+3*2;*
- *6/3;*
- *10/4;*
- *0/3;*
- *3/0;*
- *-2+100-50/2*log(exp(-2));*
- *2.3;*
- *2e10;*
- *2e+10;*
- *2e-10;*
- *2E+10;*
- *2E-10;*
- *0xDEAF;*
- *0XDEAF;*

Ejercicio C

Objetivo

El fichero de entrada podrá incorporar información con **datos identificativos** del autor, cumpliendo las siguientes especificaciones léxicas:

- Nombre y apellidos (comenzando con mayúsculas)
- Una dirección de email
- Un DNI de España

- Una matrícula de turismo español
- Una fecha, en formato dd/mm/aaaa

Cuando el analizador identifique estos campos se limitará a indicarlos por pantalla:

Nombre y apellidos: XX XX XX

e-mail: XX

DNI: XX

etc.

Si el texto de entrada no puede reconocerse deberá salir el aviso correspondiente

Solución

Se reconocen las palabras clave en base a su formato con los siguientes alcances:

- **Nombre y apellidos:** Se reconocen nombres y apellidos con la primera letra mayúscula y el resto minúsculas. Se ha diseñado de que acepte desde un nombre sin apellidos hasta n apellidos.
- **DNI:** Se reconocen códigos de DNI tanto de 7 y como de 8 caracteres (incluso empezando por 0) con o sin “-”.
- **Matrícula:** Se reconocen los 3 tipos de matrícula (con o sin “-”): las anteriores a 1971 (XX-12345), las anteriores a las actuales (XX-1234-YY) y las actuales (1234-XXX). En todos los casos se permite como válidas tanto con 1 letra como 2 (ó 3 en el caso de las actuales).
- **Fecha:** Se reconocen fechas en el formato usado en España (DD/MM/YYYY) . Se reconocen tanto casos en que se escribe el 0 delante como cuando no (06/02/2020 ó 6/2/2020).

Tokens

Ha sido necesario modificar Whitespace y crear WhitespaceNewline para separar distintos casos y así gestionar bien el espacio entre nombre y apellidos.

- *Whitespace = [\t\f]*
- *WhitespaceNewline = [\t\f] | {Newline}*
- *NombrePalabra = [A-ZÑÁÉÍÓÚ] [a-zñáéíóú]**
- *NombreApellidos = ({NombrePalabra} {Whitespace}*)+*
- *Email = .+ "@" .+ "." .+*
- *Dni = [0-9]?[0-9]{7} "-" [A-Z] | [0-9]{7,8} [A-Z]*

- *MatriculaRetro = [A-Z]{1,2} "-" [0-9]{5} | [A-Z]{1,2} [0-9]{5}*
- *MatriculaAntigua = [A-Z]{1,2} "-" [0-9]{4} "-" [A-Z]{1,2} | [A-Z]{1,2} [0-9]{4} [A-Z]{1,2}*
- *MatriculaNueva = [0-9]{4} "-" [A-Z]{1,3} | [0-9]{4} [A-Z]{1,3}*
- *Matricula = {MatriculaRetro} | {MatriculaAntigua} | {MatriculaNueva}*
- *Fecha = [0-3]? [0-9] "/" [0-1]? [0-9] "/" [0-9]{4}*

Tests

- *Juan*
- *Juan Castro Huertas*
- *Juan Castro Huertas Ruiz*
- *Juan Castro Huertas Ruiz Huecas*
- *juan@castro.com*
- *12345678-X*
- *12345678X*
- *TO-12345*
- *TO-1234-AB*
- *1234-ABC*
- *TO12345*
- *TO1234AB*
- *1234ABC*
- *02/05/2020*
- *2/5/2020*
- *31/12/2020*

3 Análisis Sintáctico

En esta sección se aborda el análisis sintáctico del lenguaje, donde se declararan las producciones donde reconocer el orden de los símbolos.

Para poder comunicar el **parser** con el lexer, el parser exporta los símbolos terminales declarados en parser.cup a la **tabla de símbolos** (que a su vez coincidirá con los del parser léxico).

Ejercicio A

Objetivo

Incorporar nuevos símbolos de la gramática y las acciones para incorporar en la gramática funciones científicas y calcular el resultado: **exp(), log ()**

En esta sección se aborda el análisis léxico del lenguaje, donde se declararan los patrones Regex con los que reconocer como tokens las palabras a reconocer del archivo input.txt.

Para poder comunicar el lexer con el parser, el lexer “exporta” los tokens reconocidos o sus alias incluyendo en los casos que sea necesario, una variable con el token reconocido y en el adecuadamente tipado. Estos tokens o alias “exportados” deben coincidir con los declarados en la tabla de símbolos (que a su vez coincidirá con los del parser sintáctico).

Basado en algunos de los ejemplos del enunciado, todas las sentencias se diseñado para esperar un cierre con “;”, excepto en el Ejercicio C, donde las palabras clave a reconocer no esperan un cierre con “;” por no estar indicado.

Solución

Se reconocen las expresiones con las operaciones matemáticas “exp(“ y “log(“ (completadas con otro “)” y cualquier expresión en medio). De esta manera, se reconocen **expresiones anidadas** dentro de estas nuevas operaciones. Tras ser reconocida, es operada usando la librería Math de Java.

Tokens

- *Exponential = "exp("*
- *Logarithm = "log("*

Tests

- *exp(2);*
- *3*exp(2.5);*
- *log(2);*

- *3*log(2.5);*
-

Ejercicio B

Objetivo

Eliminar las reglas de precedencia y ver el efecto en el analizador sintáctico. Modificar la gramática para evitar la necesidad de utilizar reglas de precedencia de la solución, incorporando los símbolos no terminales necesarios.

Además introducir el operador de cambio de signo sin necesidad de utilizar precedencia e incorporar el signo “+” también como modificador de signo.

Solución

Se han reordenado las producciones factorizando las expresiones relacionadas con operaciones matemáticas entre otras 2 nuevas producciones, **term** y **factor,** logrando así controlar la prioridad y precedencia de las operaciones sin necesidar de usar la sentencia *“precedence”*.

Además se ha introducido el símbolo “+” como antecesor a un número.

Tokens

- *N/A*

Tests

- *5*+3-80/10;*
- *10/5*2;*
- *-3*4;*
- *3-5*2;*
- *5*(4-3)/(8-(4-1));*
- *100-50/2;*

Ejercicio C

Objetivo

Incorporación de variables en memoria, de manera que los operadores participantes en las operaciones podrán ser los números literales y variables de memoria denominadas como “MEMX”, siendo X un número del 0 al 9. Por tanto, existirán dos tipos de sentencias: expresiones matemáticas y asignaciones.

Las asignaciones comienzan por el nombre de variable, “MEMX”, seguido del signo igual ‘=’ y terminado con una expresión matemática normal.

El valor por defecto de estas variables es cero, por lo que puede ser usadas en una expresión sin haber aparecido con anterioridad en la parte izquierda de una asignación.

Solución

Se ha conseguido reconocer correctamente, tanto en el reconocedor léxico como el sintáctico, los tokens y su orden, tanto referidos al nombre de variable, al símbolo de asignación, cualquier expresion asignada y reconocer el conjunto como una statement, pero no ha sido posible implementar la lógica de guardado en memoria para la su posterior reutilización.

Tokens

- *Assign = {Whitespace}* "=" {Whitespace}**
- *Variable = "MEM" {Number}*

Tests

- *MEM1 = 2;*

4 Conclusión

Esta segunda parte de la práctica 1 ha servido para afianzar en general conceptos mas avanzados sobre las producciones de una gramática para un analizador sintáctico.
En primer lugar, ha servido para afianzar la definición de gramáticas **controlando la precedencia** mediante la distinta composición de las producciones.

En segundo lugar, me ha permitido aprender a reconocer **asignaciones de variables**.

Sin embargo, no ha podido servir para aprender la correcta implementación de la lógica necesaria para guardar variables en memoria para su posterior uso.