/\*\*

This program uses the Enum type introduced in JDK 1.5.0.

This class is a lexical analyzer for the tokens defined by the grammar:

<plus> --> +

<minus> --> -

<times> --> \*

<div> --> /

<LParen> --> "("

<RParen> --> ")"

<int> --> { <digit> }+

<id> --> <letter> { <letter> | <digit> }

<float> --> { <digit> }+ "." { <digit> }+

<floatE> --> <float> (E|e) [+|-] { <digit> }+

This class implements a DFA that will accept the above tokens.

The DFA has 10 final states represented by enum-type literals:

state token accepted

Id identifiers

Int integers

Float floats without exponentiation part

FloatE floats with exponentiation part

Plus +

Minus -

Times \*

Div /

LParen (

RParen )

The DFA also uses 4 non-final states:

state string recognized

Start the empty string

Period float parts ending with "."

E float parts ending with E or e

EPlusMinus float parts ending with + or - in exponentiation part

The states are represented by an Enum type called "State".

The function "driver" is the driver to operate the DFA.

The function "nextState" returns the next state given

the current state and the input character.

To modify this lexical analyzer to recognize a different token set,

the functions "nextState", "isFinal" and the enum type "State" need to be modified;

the function "driver" and the other utility functions remain the same.

\*\*/

import java.io.\*;

public abstract class lexArith

{

public enum State

{

// non-final states ordinal number

Start, // 0

Period, // 1

E, // 2

EPlusMinus, // 3

// final states

Id, // 4

Int, // 5

Float, // 6

FloatE, // 7

Plus, // 8

Minus, // 9

Times, // 10

Div, // 11

LParen, // 12

RParen, // 13

UNDEF

}

// By enumerating the non-final states first and then the final states,

// test for a final state can be done by testing if the state's ordinal number

// is greater than or equal to that of Id.

public static String t; // holds an extracted token

public static State state; // the current state of the FA

private static int a; // the current input character

private static char c; // used to convert the variable "a" to

// the char type whenever necessary

private static BufferedReader inStream;

private static PrintWriter outStream;

private static int getNextChar()

// Returns the next character on the input stream.

{

try

{

return inStream.read();

}

catch(IOException e)

{

e.printStackTrace();

return -1;

}

} //end getNextChar

private static int getChar()

// Returns the next non-whitespace character on the input stream.

// Returns -1, end-of-stream, if the end of the input stream is reached.

{

int i = getNextChar();

while ( Character.isWhitespace((char) i) )

i = getNextChar();

return i;

} // end getChar

private static int driver()

// This is the driver of the FA.

// If a valid token is found, assigns it to "t" and returns 1.

// If an invalid token is found, assigns it to "t" and returns 0.

// If end-of-stream is reached without finding any non-whitespace character, returns -1.

{

State nextState; // the next state of the FA

t = "";

state = State.Start;

if ( Character.isWhitespace((char) a) )

a = getChar(); // get the next non-whitespace character

if ( a == -1 ) // end-of-stream is reached

return -1;

while ( a != -1 ) // while "a" is not end-of-stream

{

c = (char) a;

nextState = nextState( state, c );

if ( nextState == State.UNDEF ) // The FA will halt.

{

if ( isFinal(state) )

return 1; // valid token extracted

else // "c" is an unexpected character

{

t = t+c;

a = getNextChar();

return 0; // invalid token found

}

}

else // The FA will go on.

{

state = nextState;

t = t+c;

a = getNextChar();

}

}

// end-of-stream is reached while a token is being extracted

if ( isFinal(state) )

return 1; // valid token extracted

else

return 0; // invalid token found

} // end driver

private static State nextState(State s, char c)

// Returns the next state of the FA given the current state and input char;

// if the next state is undefined, UNDEF is returned.

{

switch( state )

{

case Start:

if ( Character.isLetter(c) )

return State.Id;

else if ( Character.isDigit(c) )

return State.Int;

else if ( c == '+' )

return State.Plus;

else if ( c == '-' )

return State.Minus;

else if ( c == '\*' )

return State.Times;

else if ( c == '/' )

return State.Div;

else if ( c == '(' )

return State.LParen;

else if ( c == ')' )

return State.RParen;

else

return State.UNDEF;

case Id:

if ( Character.isLetterOrDigit(c) )

return State.Id;

else

return State.UNDEF;

case Int:

if ( Character.isDigit(c) )

return State.Int;

else if ( c == '.' )

return State.Period;

else

return State.UNDEF;

case Period:

if ( Character.isDigit(c) )

return State.Float;

else

return State.UNDEF;

case Float:

if ( Character.isDigit(c) )

return State.Float;

else if ( c == 'e' || c == 'E' )

return State.E;

else

return State.UNDEF;

case E:

if ( Character.isDigit(c) )

return State.FloatE;

else if ( c == '+' || c == '-' )

return State.EPlusMinus;

else

return State.UNDEF;

case EPlusMinus:

if ( Character.isDigit(c) )

return State.FloatE;

else

return State.UNDEF;

case FloatE:

if ( Character.isDigit(c) )

return State.FloatE;

else

return State.UNDEF;

default:

return State.UNDEF;

}

} // end nextState

private static boolean isFinal(State state)

{

return ( state.compareTo(State.Id) >= 0 );

}

public static void getToken()

// Extract the next token using the driver of the FA.

// If an invalid token is found, issue an error message.

{

int i = driver();

if ( i == 0 )

displayln(t + " -- Invalid Token");

} // end getToken

public static void display(String s)

{

outStream.print(s);

}

public static void displayln(String s)

{

outStream.println(s);

}

public static void setLex(String inFile, String outFile)

// Sets the input and output streams to "inFile" and "outFile", respectively.

// Also sets the current input character "a" to the first character on

// the input stream.

{

try

{

inStream = new BufferedReader( new FileReader(inFile) );

outStream = new PrintWriter( new FileOutputStream(outFile) );

a = inStream.read();

}

catch(FileNotFoundException e)

{

e.printStackTrace();

}

catch(IOException e)

{

e.printStackTrace();

}

} // end setIO

public static void closeIO()

{

try

{

inStream.close();

outStream.close();

}

catch(IOException e)

{

e.printStackTrace();

}

} // end closeIO

public static void main(String argv[])

// The input/output file names must be passed as argv[0] and argv[1].

{

int i;

setLex( argv[0], argv[1] );

while ( a != -1 ) // while "a" is not end-of-stream

{

i = driver(); // extract the next token

if ( i == 1 )

displayln( t+" : "+state.toString() );

else if ( i == 0 )

displayln( t+" -- Invalid Token");

}

closeIO();

} // end main

}