221 Compilers – Exercise 1: writing a very tiny interpreter

The object of this exercise is to write an interpreter in Java for a very simple programming language—in fact a language for controlling the motions of a toy robot or "turtle", which draws on the floor as it moves around. Note that this exercise introduces examinable material not covered in the lecture notes.

This is a pencil-and-paper exercise but afterwards you might like to play with the code, which you can find at

`http://www.doc.ic.ac.uk/~phjk/Compilers/SampleCode/`

You need to read and understand around 300 lines of Java, then fill in the gaps.

The source language

To get an idea of the language here is a small example program:

```
program "Sample"
begin
  forward 20 ;
  times 3 do
  begin
    turn 108 degrees ;
    forward 10 ;
  end ;
end
```

The syntax is specified in BNF as follows:

program	→	'`program`' string statement
statement	→	'`turn`' number '`degrees`' |
		'`forward`' number |
		'`times`' number '`do`' statement |
		'`begin`' statement-list
statement-list	→	'`end`' |
		statement '`;`' statement-list

Notice that terminals (such as '`program`') appear in turtle programs generated from the grammar, while non-terminals like 'program', 'statement' and 'statement-list' do not. The non-terminal "string" stands for any sequence of printable characters demarcated by quotes "`"`". The non-terminal "number" stands for a sequence of digits. We will handle strings and numbers in the lexical analyser (the method `nextSymbol` introduced later deals with them).

Exercise 1.1: The parse tree

What to do

- Prove that the example program above is syntactically correct by drawing out the parse tree—that is, use the BNF rules to derive the example program. You need to do this quickly as there is plenty more to do! Once you have the idea, move on.

Exercise 1.2: The Interpreter

What to do

Read the Java code which follows. Fill in the gaps, which are marked with `??` in the left margin. Given the sample turtle program above, the interpreter is supposed to print the following:

```
Please move forward 20
Please turn 108 degrees
Please move forward 10
Please turn 108 degrees
Please move forward 10
Please turn 108 degrees
Please move forward 10
```

Your job is to make it so.

The code is presented in a logical order - starting with lexical tokens, then parsing, then the abstract syntax tree, and finally the interpreter. I suggest you skim the Tokens and Lexer code and start with the Turtle class, especially the "parse" functions.

Tokens.java

```
class Token {
    static final int PROGRAM   = 0;
    static final int STRING    = 1;
    static final int TURN      = 2;
    static final int DEGREES   = 3;
    static final int FORWARD   = 4;
    static final int TIMES     = 5;
    static final int DO        = 6;
    static final int NUMBER    = 7;
    static final int BEGIN     = 8;
    static final int END       = 9;
    static final int SEMICOLON = 10;

    int tokenId;     // which token was it - see above list
    String string;   // string rendering of token
    int intValue;    // if token is NUMBER this gives number's value

    Token(String s) {  // Token constructor converts string to token
        string = s;
        if (s.equals("program"))      tokenId = PROGRAM;
        else if (s.equals("turn"))    tokenId = TURN;
        else if (s.equals("degrees")) tokenId = DEGREES;
        else if (s.equals("forward")) tokenId = FORWARD;
        else if (s.equals("times"))   tokenId = TIMES;
        else if (s.equals("do"))      tokenId = DO;
        else if (s.equals("begin"))   tokenId = BEGIN;
        else if (s.equals("end"))     tokenId = END;
        else if (s.equals(";"))       tokenId = SEMICOLON;
```

```
        else if (s.startsWith("\"") && s.endsWith("\""))
            tokenId = STRING;
        else try {
                intValue = Integer.parseInt(s.trim());
                tokenId = NUMBER;
            } catch (NumberFormatException nfe) {
                System.out.println("Unrecognised token "+s);
            }
    }
    static String name(int tokenId) {
        switch(tokenId) {
        case PROGRAM:   return("program");
        case TURN:      return("turn");
        case DEGREES:   return("degrees");
        case FORWARD:   return("forward");
        case TIMES:     return ("times");
        case DO:        return ("do");
        case BEGIN:     return ("begin");
        case END:       return ("end");
        case SEMICOLON: return (";");
        case NUMBER:    return ("number");
        default:
            return ("UNRECOGNISED_TOKEN");
        }
    }
}
```

Lexical analysis: Lexer.java

```
import java.io.*;

class Lexer {
    DataInputStream inStream;
    Token currentToken;
    boolean replacedTokenExists;
    Token replacedToken;

    Lexer(DataInputStream in) {
        replacedTokenExists = false;
        replacedToken = null;
        inStream = in;
    }

    /** Read next symbol/keyword, return as Token.
     * Remember what it was in case we want to inspect it again.
     * If a token has been pushed back onto the input, return that instead
     */
    Token nextToken() throws IOException {
        if (replacedTokenExists) {
```

```
        replacedTokenExists = false;
        currentToken = replacedToken;
        return replacedToken;
    } else {
        String word = getWord();
        currentToken = new Token(word);
        return currentToken;
    }
}
Token getLastToken() {
    if (currentToken == null) {
        System.out.println("currentToken is null");
    }
    return currentToken;
}
void replaceToken(Token t) {
    if (replacedTokenExists) {
        System.out.println("replaceToken() called twice, token lost");
    }
    replacedTokenExists = true;
    replacedToken = t;
}
void match(int expectedTokenId) throws IOException {
    Token t = nextToken();
    if (t.tokenId != expectedTokenId) {
        System.out.println("Syntax error: "+Token.name(expectedTokenId)+
                           " expected, but "+t.string+" found");
    }
}
/**
 * Get the next space(etc)-separated word/number from the stream
 */
String getWord()
    throws IOException
{
    String theWord = new String();
    char ch = (char) inStream.readByte();
    // skip blanks and comment lines
    while (IsSkippable(ch)) {
        ch = (char) inStream.readByte();
    }
    while(!IsSkippable(ch)) {
        theWord = theWord+ch;
        ch = (char) inStream.readByte();
    }
    return theWord;
}
private static boolean IsSkippable(char ch) {
    return (ch == ' ') || (ch == '#') || (ch == '\n')
        || (ch == '\r') || (ch == '\t');
```

```
    }
}
```

The parser: Turtle.java

```
import java.io.*;

public class Turtle {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.out.println("Usage: java Turtle inputfile"); System.exit(0);
        }
        DataInputStream in = null;
        try {
            in = new DataInputStream(new FileInputStream(args[0]));
            Lexer lex = new Lexer(in);
            parseProgram(lex);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }

    static void parseProgram(Lexer lex) throws IOException {

        lex.match(Token.PROGRAM);
        Token t = lex.nextToken();
        if (t.tokenId == Token.STRING) {
            System.out.println("Program name: "+t.string);
        } else {
            System.out.println("Syntax error: program name string expected");
        }

        StatementTree result= parseStatement(lex);

        result.Accept(new InterpretVisitor());
    }

    static StatementTree parseStatement(Lexer lex) throws IOException {
        Token t = lex.nextToken();
        switch (t.tokenId) {
        case Token.TURN:
            lex.match(Token.NUMBER);
            int degrees = lex.getLastToken().intValue;
            lex.match(Token.DEGREES);
            return new TurnNode(degrees);

        case Token.FORWARD:
??
```

```
??
??

        case Token.TIMES:
??
??
??
??
??

        case Token.BEGIN:
            StatementTreeList tl = parseStatementList(lex);
            return new BeginNode(tl);

        default:
            System.out.println("Syntax error: statement expected, "+t.string+
                                   " found");
            return null;
        }
    }

    static StatementTreeList parseStatementList(Lexer lex) throws IOException {
        Token t = lex.nextToken();

        switch (t.tokenId) {
        case Token.END:
            return null; // nothing more to look for
        default:
            // if it's not the end, assume there's another stat

            lex.replaceToken(t);
            StatementTree first = parseStatement(lex);
            lex.match(Token.SEMICOLON);
            StatementTreeList rest = parseStatementList(lex);
            return new StatementTreeList(first, rest);
        }
    }

}
```

The Abstract Syntax Tree data type

StatementTree.java

```
public abstract class StatementTree {
    public abstract void Accept(TreeVisitor v);
}
```

StatementTreeList.java

```
public class StatementTreeList {
    StatementTree first;
    StatementTreeList rest;

    StatementTreeList(StatementTree f, StatementTreeList r) {
        first = f;
        rest = r;
    }
    public void Accept(TreeVisitor v) {
        v.visitStatementList(first, rest);
    }
}
```

TurnNode.java

```
public class TurnNode extends StatementTree {
    int degrees;
    TurnNode(int d) { degrees = d; }

    public void Accept(TreeVisitor v) {
        v.visitTurnNode(degrees);
    }
}
```

ForwardNode.java

```
public class ForwardNode extends StatementTree {
    int distance;
    ForwardNode(int d) { distance = d; }

    public void Accept(TreeVisitor v) {
        v.visitForwardNode(distance);
    }
}
```

TimesNode.java

```
public class TimesNode extends StatementTree {
    int count;
    StatementTree body;
    TimesNode(int d, StatementTree b) { count = d; body = b; }

    public void Accept(TreeVisitor v) {
        v.visitTimesNode(count, body);
    }
}
```

BeginNode.java

```
public class BeginNode extends StatementTree {
    StatementTreeList body;
    BeginNode(StatementTreeList b) { body = b; }

    public void Accept(TreeVisitor v) {
        v.visitBeginNode(body);
    }
}
```

TreeVisitor.java

```
public abstract class TreeVisitor {
    abstract void visitStatementList(StatementTree first,
                            StatementTreeList rest);
    abstract void visitTurnNode(int degrees);
    abstract void visitForwardNode(int distance);
    abstract void visitTimesNode(int count, StatementTree body);
    abstract void visitBeginNode(StatementTreeList body);
}
```

InterpretVisitor.java

```
public class InterpretVisitor extends TreeVisitor {
    void visitStatementList(StatementTree first,
                            StatementTreeList rest) {
        first.Accept(this);
        if (rest != null) {
            rest.Accept(this);
        }
    }
    void visitTurnNode(int degrees) {
        System.out.println("Please turn "+degrees+" degrees");
    }
    void visitForwardNode(int distance) {
??
    }
    void visitTimesNode(int count, StatementTree body) {
??
??
??
    }
    void visitBeginNode(StatementTreeList body) {
        body.Accept(this);
    }
}
```

Paul Kelly, Imperial College London (2012)