Assignment 6: The ML Module System and Object-Oriented Programming in Eiffel

CS 442/642

Due April 8^{th}

For this assignment, you will use Standard ML of New Jersey and the SmartEiffel 1.1 implementation of the Eiffel programming language.

Part A—The ML Module System

Pages 153–155 of your course notes give an implementation of an interpreter for deterministic finite automata (DFAs) using ML functors. For this portion of the assignment, your first task is to design a similar interpreter for *nondeterministic* finite automata (NFAs).

1. An NFA interpreter

Recall that nondeterministic finite automata differ from their deterministic counterparts by offering a choice of 0 or more alternative next states for each given (state, input) pair. Such a machine accepts if there exists a set of choices that land the machine in an accepting state when the input is consumed.

Since ML is not a nondeterministic language, we will simulate the action of the NFA by keeping track of the *set* of states the automaton could be in after having read a given sequence of characters. If, after having consumed the input, at least one of the machine's possible configurations is an accepting state, the machine accepts.

For this task, you will modify the DFA signature in the course notes so that it now specifies the characteristics of an NFA. You will then write a functor `NFA` with the following interface:

```
functor NFA (Spec : NFASIG) : sig
   val run : Spec.Sigma list -> bool
end
```

This functor accepts a specification for an NFA and outputs a structure that provides a single function `run` that determines whether a given input is in the language specified by the NFA.

2. DFAs are NFAs

Since every DFA is implicitly an NFA, it would be useful if your NFA functor could accept and interpret DFA specifications. To make this possible, you will write a functor called `DFAtoNFA` that takes a DFA specification and outputs an equivalent NFA specification. (Note: since every DFA is implicitly an NFA, this functor should be *very* simple.)

3. Building Blocks

One of the nice features of regular languages is that they are closed under many operations. As a result, we have many tools at our disposal for creating new regular languages from old ones. In particular, we

may specify a regular language as the intersection of two other regular languages. For this task, you will use closure under intersection as the basis for a tool to build new DFAs from old ones. In particular, you will write a functor called `Intersection`, parameterized by two structures called `Spec1` and `Spec2`, with signature `DFASIG`, and returning a structure with signature `DFASIG` that recognizes the intersection of the langauges of the two input automata.

4. Discussion

1. What problem would we run into if we tried to write a functor to compute a DFA that recognizes the *union* of the languages of two input automata? How could we modify `DFASIG` to fix the problem?

Notes

1. The purpose of this portion of the assignment is to explore a useful application of the ML module system, and not to examine how much you remember from automata theory. Therefore, if you need a refresher, feel free to come to office hours to discuss some of the necessary constructions. You are also permitted to discuss DFA constructions on Piazza.

Part B—Setup

Three files have been made available for download:

- `http://www.student.cs.uwaterloo.ca/~cs442/tokenizer.e`
- `http://www.student.cs.uwaterloo.ca/~cs442/tokenizerclient.e`
- `http://www.student.cs.uwaterloo.ca/~cs442/test.pl`

The class `TOKENIZER` implements a simple tokenizer for variable-free Prolog facts and rules. The class `TOKENIZERCLIENT` is a "main" class that uses `TOKENIZER`. Download all three files. Then, with your account set up properly to run SmartEiffel, type the following to compile:

```
compile -o tokenkizerclient tokenizerclient.e
```

Then run the program as follows:

```
./tokenizerclient test.pl
```

You should see the following output:

```
a
:-
b
c
d
.
e
:-
f
.
g
.
abc
:-
de
```

```
fg
hi
.
EOF
```

Each line in the output contains a single lexical token from the file `test.pl`.

For this assignment, you will implement Prolog's backtracking search strategy for answering queries. Your task is greatly simplified by the absence of variables, so unification and substitutions are not needed.

Part 1

Define classes for facts, rules, and the database. Depending on your design decisions, you may or may not have exactly one class for each of these elements; you may have more or you may have fewer. Write code to fetch tokens from the provided tokenizer and populate a database object with the corresponding facts and rules. Remember that the order in which entries appear in the database is significant.

Part 2

Define a class or classes to represent a Prolog search tree. Use the examples in Chapter 8 of your course notes to guide your design as you see fit.

Part 3

Write code to traverse/construct your search tree in answer to a Prolog query. The query will be passed in on the command line. For each successful match to the query, your program should list the sequence of rule applications that produced the answer. For example, given the database and query on p. 167 of your course notes, your program should output

```
2 4 5 4
6 5 4
```

to the screen. For the example on p. 168, your program should output an infinite sequence of

```
2 4 5 4
6 5 2 4 5 4
6 5 6 5 2 4 5 4
. . .
```

(each line represents the sequence of matches resulting in a single success). For the example on p. 169, your program should run forever (or until it runs out of memory) without producing any output.

Part 4

Once you have your backtracking search programmed and working, add support for the cut. Recall that the cut, if reached, prevents alternatives in the levels of the tree that contain the cut, and one above, from being tried. Running your solution on the example given on pp. 171–172 should produce the following output:

```
9 1 2 4 5 7
9 1 2 4 6
10
```

Notes

- Use the tree diagrams in the course notes as a model, and structure your program that way. If you follow the procedure outlined there, your implementation should be straightforward.
- Adding the cut to a working program, if implemented as described in the notes, should require only small changes.

Electronic Submission

For part A, submit your solution to the entire programming portion as the single file `a6.sml`. For part B, submit your solution to the entire programming portion as a single program. Your design and decomposition of the problem into classes will dictate the files you need to submit. However, your "main" class should be called `A6`, and be stored in the file `a6.e`. The TA will use the following command to compile your program:

```
compile -o a6 a6.e
```

The first argument on the command line for your program represents the name of the file that contains the database to be used. All remaining command line arguments constitute the query. For example, the following command line

```
./a6 db.pl a c e
```

invokes your program on the database specified in `db.pl` and answers the query `?- a,c,e.`. Your program will output to the screen the result of running the query on the database.

Paper Submission

Your paper submission will include:

- all source code;
- answer to the discussion question in part A.

The distribution of grades will be 40% for part A, and 60% for part B.