CS 241 — Winter 2022 — Assignment 4

Assignments for CS 241

-->← Assignment 3	Assignment 4	Assignment 5 →
Friday, Feb 4h at 5:00 pm	**Friday, Feb 11th at 5:00 pm**	Friday, Feb 18th at 5:00 pm

P1 • P2 • P3 • P4 • P5 • P6 • P7 • P8

In this assignment, you will practice creating DFAs and write a DFA recognizer. You will also learn the basics of WLP4, the programming language you will write a compiler for in this course.

Marmoset Notes

Problems 1 through 4 have a single public test *only*, no release tests or secret tests.

For this assignment, secret tests are worth **10%** of the Marmoset test marks.

Part I. DFAs

For Problems 1 to 4, you are to define DFAs (deterministic finite automata) to recognize several formal languages. Each DFA is represented as a file that lists the alphabet, states, initial state, accepting states, and transitions that comprise the DFA, in the format described in http://www.student.cs.uwaterloo.ca/~cs241/dfa/DFAfileformat.html. Your submission to Marmoset will be a text file in this format, containing the description of a DFA.

For testing, an implementation of Problem 5, which reads a file in the DFA File Format and implements the DFA recognizer, is available in the student.cs environment. To use it, enter

```
cs241.DFA < inputfile
```

where *inputfile* is a DFA in the DFA File Format, plus a set of test inputs as described in Problem 5.

Problem 1 — 10 marks of 100 (filename: `notdiv3.dfa`)

Write a DFA with alphabet {0,1} that recognizes the language of binary integers that have no useless leading zeroes and are not divisible by 3. The first few such integers are 1, 10, 100, 101, 111, ...

Problem 2 — 10 marks of 100 (filename: `not23.dfa`)

Write a DFA with alphabet {0,1} that recognizes the language of binary integers that have no useless leading zeroes, are not divisible by ~~2, a~~nd are not divisible by 3. The first few such integers are 1, 101, 111, 1011, 1101, ...

Problem 3 — 10 marks of 100 (filename: `int.dfa`)

Write a DFA that recognizes the language of decimal integers between -128 and 127 inclusive, with no useless leading zeroes. Your DFA should recognize *integers*, not expressions involving integers and the unary minus operator, so things like `-0` or `--12` should not be accepted.

The alphabet symbols are {0,1,2,3,4,5,6,7,8,9,- }.

Problem 4 — 10 marks of 100 (filename: `pets.dfa`)

We normally think of "strings" as consisting of individual "characters". But in formal language theory, a "string" or "word" is just a finite sequence of alphabet symbols. "Symbols" of an alphabet might consist of multiple characters, and the DFA file format supports this.

Write a DFA with alphabet {cat, dog, iguana, mouse}. It should recognize a sequence of these alphabet symbols if and only if it contains the following **contiguous** subsequence: *cat dog iguana cat dog mouse*. For example, *cat mouse* ***cat dog iguana cat dog mouse*** *cat dog* should be accepted.

Problem 5 — 25 marks of 100, 8 secret marks (filename: `dfa.rkt` or `dfa.cc`)

Write a Racket or C++ program that implements a DFA recognizer. Your program should read (from standard input) a DFA description, followed by several additional lines of input. Each additional line of input is a (possibly empty) sequence of alphabet symbols separated by spaces. For each sequence, your program should print "true" if the word formed from the alphabet symbols is in the language, and "false" if it is not. You may assume that the input to your program is valid according to the definition given in the DFA File Format specification.

Warning: Reading input in C++ has a lot of pitfalls. If your knowledge of C++ streams is rusty, see this brief review.

Sample input:

```
2
0
1
5
start
zero
0mod3
1mod3
2mod3
start
2
zero
0mod3
8
start 0 zero
start 1 1mod3
1mod3 0 2mod3
1mod3 1 0mod3
2mod3 0 1mod3
2mod3 1 2mod3
0mod3 0 0mod3
0mod3 1 1mod3

0
1
1 0
1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

Correct output for sample input:

```
false
true
false
false
true
false
false
true
false
```

Note that there are *nine* input strings here, and nine corresponding outputs. The blank line represents giving the empty string as input.

Part II. The WLP4 Programming Language

The WLP4 Programming Language is essentially a (very) small subset of C++. In this assignment, you will learn the basics of WLP4 and construct a DFA that recognizes WLP4 tokens. Starting from the next assignment, you will begin writing the components of a compiler that translates WLP4 to MIPS assembly language!

As an example, the following WLP4 program computes the sum of two integers, a and b.

```
//
// WLP4 program with two integer parameters, a and b
//   returns the sum of a and b
//
int wain(int a, int b) {
    return a + b;   // unhelpful comment about summing a and b
}
```

You may test this program on the Linux environment by placing it in a file named `test.wlp4` and entering the following commands, which compile it to MIPS machine language and run it with the familiar `mips.twoints` command from Assignments 1 and 2:

```
wlp4c < test.wlp4 > test.mips
mips.twoints test.mips
```

Note that the "wain" function (WLP4's version of "main") in the above program takes two integers as parameters. These are specified by entering values for $1 and $2 when prompted by `mips.twoints`. The return value is stored in $3.

You can write WLP4 programs which take arrays as input as follows. Change the first parameter of the "wain" function to type `int*`, then run the compiled MIPS code with `mips.array`. The first parameter will be a pointer to the start of the input array, and the second parameter will be the size of the array.

Before doing the following problems, please read the WLP4 Programming Language Tutorial which highlights some of the key differences between WLP4 and C++.

There are many unusual restrictions in WLP4. If you just start writing normal C++ code, it likely will not work and you will have to waste a lot of time revising it.

The WLP4 Language Specification should be consulted for the definitive specification of the WLP4 language.

Problem 6 — 5 marks of 100, 1 secret mark (filename: `exp.wlp4`)

It is sometimes useful to compute x^n, but the naive approach requires n multiplications, which can be unacceptably inefficient. Instead, for fixed-sized numbers, exponentiation can be achieved in O(log(n)) time, where n is the exponent, using a method called exponentiation by squaring. Write a WLP4 program that takes two parameters: x and n, and returns x^n, where x and n are integers, and n is non-negative, using the exponentiation by squaring method. **Your program must solve the problem using iteration, not recursion.**

Hint: You are allowed to use the page https://en.wikipedia.org/wiki/Exponentiation_by_squaring as a reference.

Problem 7 — 5 marks of 100, 1 secret mark (filename: `binsearch.wlp4`)

Write a WLP4 **function** that performs binary search on an array of integers. The function must be called `binsearch` and must accept 3 parameters: a pointer to the beginning of an array of integers, an integer representing the array's size, and an integer value to search for in the array, in that order. You may assume that the array is sorted in ascending order and contains each unique value exactly once. If the search value is in the array, your function should return its index. If it is not, your function should return -1.

To test the function, you will of course need to write a `wain` function as well, as WLP4 programs will not compile without a `wain` function. **However, you are to submit only the function `binsearch`; the file you submit should not contain a `wain` function.** We will use our own `wain` function to test your implementation.

Hint: You are allowed to use the page https://en.wikipedia.org/wiki/Binary_search_algorithm as a reference.

Problem 8 — 25 marks of 100 (filename: `wlp4.dfa`)

Using the DFA File Format, create a deterministic finite automaton that recognizes the language of valid WLP4 tokens from the list given in the "Lexical Syntax" section of the WLP4 specification, with the exception that you do not need to do bounds checking on NUM tokens (see the "Note on Numbers" below).

The alphabet of the DFA should consist of every individual ASCII character that may appear in a token. There are 80 such characters. Note that this does not include white space characters, since white space (including comments) is not considered a kind of token by the specification.

Your DFA should recognize *individual tokens*, not sequences of tokens. Recognizing sequences of tokens is the job of a *scanner*. You are not writing a scanner (yet), you are simply creating a DFA.

Additionally, you are not required to "distinguish" between different kinds of tokens. For example, you do not necessarily need a different accepting state for each keyword. The only requirement for this question is that your DFA *recognizes the language of valid tokens*. There are no restrictions on "how" it recognizes them. When you write a *scanner*, you might find that "how" the DFA recognizes tokens affects the design of the scanner, but you are not writing a scanner (yet). We recommend trying to solve this problem with the simplest DFA you can think of, and worry about the scanner later.

Some examples of input strings you should accept are: `else` (a keyword), `elsa` (an identifier), `241` (a number), `!=` (an operator), `;` (punctuation).

Some examples of input strings you should not accept are: the empty string, `!` (not a valid token), `x+(y-1)` (sequence of tokens), `int i = 0;` (sequence of tokens including whitespace), `$` (not even part of the alphabet). (Note: you won't actually be tested with strings containing characters that are not part of the alphabet.)

Note on Numbers: The WLP4 specification states that a NUM token's value may not exceed 2^{31}-1. If you did the earlier DFA problems, you know how annoying it is to check that a number falls within a specific range using a DFA. Thus, we do not require you to perform bounds checking in your DFA. You can simply accept all strings of digits with no useless leading zeroes. That said, you will not be punished on Marmoset for doing bounds checking if it is done correctly, but it is not required, and will make your DFA more complicated.

Unlike the previous DFA problems (Problems 1 through 4), this problem has release tests. However, there are no secret tests for this problem.

Click here to go back to the top of the page.