Introduction to Theoretical Computer Science

Coursework 1

due 12:00 on Friday 27th October 2023

All questions except for question 3 are written questions. If you wish, you may typeset your answers, but you may also submit **clean scans** of manuscript. The mechanism for electronic submission will be announced by a week before the hand-in date.

This coursework is formative, and no marks will be assigned.

1 Regular languages

Recall that in lectures we showed that the class of regular languages was closed under union, sequential composition, and Kleene closure.

(a) The complement of a language L, written $\overline{L}$, is every string not in L, i.e. $\Sigma^* \setminus L$. Show that the regular languages are closed under complement.

(b) Hence or otherwise, show that the regular languages are closed under intersection. The intersection of two languages $L_1 \cap L_2$ is the set of all strings that are in both L_1 and L_2, that is $\{w \mid w \in L_1 \land w \in L_2\}$.

(c) An *all-NFA* is a variant of an NFA where a string w is only accepted if *all* states reached on word w are final, i.e. $\delta^*(q_0, w) \subseteq F$. Show that there is an all-NFA that recognises a language if and only if it is regular.

(d) Prove that $L = \{\mathtt{0}^n\mathtt{1}^m\mathtt{2}^{m-n} \mid m \geq n \geq 0\}$ is not regular. You may use any of the three methods used in lectures: the Pumping Lemma, the Myhill-Nerode theorem, or using (or proving) closure properties of the regular languages to reduce the problem to a known non-regular language. As an extension exercise, you may wish to try multiple methods.

2 Context-free languages

Consider the CFG G:

$$S \to \mathtt{a}S \mid \mathtt{a}S\mathtt{b}S \mid \varepsilon$$

(a) Informally characterise $\mathcal{L}(G)$.

(b) Show that it is *ambiguous* by finding a string for which you can construct two parse trees and two leftmost derivations.

(c) Find an unambiguous grammar for $\mathcal{L}(G)$.

(d) Give a push-down automaton that recognises $\mathcal{L}(G)$.

(e) Some $w \in \{\mathtt{a}, \mathtt{b}\}^*$ have unique parse trees in G. What are those strings w? Give an efficent test to tell whether w has this property. The test "try all parse trees to see how many yield w" is not adequately efficient.

3 Implementing Register Machines

This question has two parts:

(a) Implement a simulator for register machines. You may use the language of your choice, provided that the specification is met. A scripting language such as Perl or Python will probably be quickest.

(b) Write a Register Machine program to compute squares.

The remainder of this text gives the necessary information for the two parts.

3.1 Program behaviour specification

The executable program should be called `rmsim`. When executed, it should read a machine specification (as below) on standard input, and then print the final register contents. Optionally, if given the `-t` command-line flag, it should print a trace of the execution of the machine. **PLEASE NOTE:** you must submit an *executable* program called `rmsim`. Before submitting, check that the following command:

```
echo registers 1 | ./rmsim
```

gives the output

```
registers 1
```

3.2 Input syntax

In the following BNF-style specification, literal characters are in '`typewriter`' in quotes, *SP* means a sequence of one or more space or tab characters, *NL* means a newline character (i.e. ASCII linefeed), *number* means a sequence of digits, and *identifier* means a sequence of letters and digits starting with a letter. The input is case-sensitive. $(), ?, *, +$ have their usual regexp meanings.

$$
\begin{array}{rcl}
\textit{input} & := & \textit{regSpec NL program} \\
\textit{regSpec} & := & \text{`}\texttt{registers}\text{'}\ (\textit{SP number})* \\
\textit{program} & := & (\textit{labInst NL})* \\
\textit{labInst} & := & (\textit{label SP}?\ \text{`:'})?\ \textit{SP}?\ \textit{inst} \\
\textit{label} & := & \textit{identifier} \\
\textit{inst} & := & \quad \text{`}\texttt{inc}\text{'}\ \textit{SP register} \\
 & & \mid \text{`}\texttt{decjz}\text{'}\ \textit{SP register SP label} \\
\textit{register} & := & \text{`}\texttt{r}\text{'}\ \textit{number}
\end{array}
$$

In addition, to allow comments in programs, your program should ignore completely any line beginning with `#`.

Question continues

3.3 Input semantics

The `registers` line gives the initial values of the registers, in order from register zero up. Any other registers used by the program should be initialized with zero.

The remaining lines are the program, with lines implicitly numbered from zero. The optional *identifier*: at the start of a line is a line label. It is an error to define the same label twice, or for a program to use an undefined label, except for the special identifier `HALT`, which causes a halt if branched to. The instructions are as in the lectures, where `r`n means register n. (Errors may be detected at 'compile time', or at 'run time', as you prefer.)

3.4 Output syntax

The output should say
`registers`
followed (on the same line) by the space-separated values of the registers, from register zero up to the highest register used (including any (implicitly zero) intervening registers that are not used or mentioned).

Your program should produce no other output to standard output (unless `-t` *is given); you are free to print anything you like to standard error.*

The syntax of the tracing output is not defined; use whatever you think looks most useful.

3.5 Example

The following input

```
registers 10 5
loop:  decjz r1 HALT
       decjz r0 HALT
       decjz r2 loop
```

should produce the output
`registers 5 0 0`

3.6 A program

Once you are happy with your simulator, **write an RM program** to compute the square of a number. Write the program in the file `square.r`. This program should *not* contain an initial `registers` line; it should expect to find its input x in `r0`, and it should leave the answer x^2 in `r0`.

3.7 Submission

For this question, you should tar up your program *executable* (an executable or script that runs on DICE), and your program *source* (if not using a script), together with a `README` file if you have any comments you wish to make (including compilation instructions if you are not using an interpreted language), and your `square.r` into a file `rmsim.tar`, and submit this file. The mechanism for submission will be announced by a week before the hand-in date.

3.8 Optional extension

If you enjoy this sort of thing, and have time to spare, design and implement a macro facility along the lines of the one we used informally in lectures. Discuss any design decisions that we skated over in lectures.

In these questions, your answers should give convincing proofs at a high level, such as used on the lecture slides; you do not have to give detailed formal encodings of machines.

4 Reductions for undecidability

In lectures, we considered the Halting Problem H, the Looping Problem L, and the Uniform Halting Problem UH. Now we consider the Universal Looping problem UL: given a machine M, does M loop on all inputs R?

(a) Show, by reduction from L, that UL is undecidable.

(b) Show, by constructing a suitable machine, that UL is co-semi-decidable. (*Hint:* interleaving.)

We often (always?) want to know whether a program, or even just a function/method in a program, correctly implements its specification. Can we write programs to find this out?

Recall that we say a machine computes a function f if, when started with n in R_0, it halts with $f(n)$ in R_0.

Take f to be the factorial function $f(n) = n!$.

Let the decision problem *Fac* be the (codes of) the register machines that compute f.

(c) **Construct** a reduction from H to *Fac*, and so show that *Fac* is undecidable.

Hint: you need to start with an arbitrary program, for which we want to know whether it halts, and end up with an 'is it a factorial function?' problem. Probably you won't much care what the arbitrary program actually computes ...

Thus we can't write programs to check that other programs do anything interesting at all! (There was nothing very special about the factorial function.) Remember to show that your reduction *is* a reduction, according to the definition in lectures.