Introduction to Theoretical Computer Science
Coursework 1
due 16:00 Friday 17 February 2017

Each question carries 10 marks; questions are not necessarily of equal length or difficulty. I expect each question to take around two hours, or a little more for the first. It would be useful to me if you would note on each question how long you spent on it – in terms of actual working time, excluding digressions on to Facebook etc.!

1 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. [6]
(b) Write a Register Machine program to compute triangle numbers. [4]
The remainder of this text gives the necessary information for the two parts.

1.1 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
```

1.2 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.

```
input := regSpec NL program
regSpec := 'registers' (SP number)*
program := (labInst NL)*
labInst := (label SP? ':')? SP? inst
label := identifier
inst := 'inc' SP register
       | 'decjz' SP register SP label
register := 'r' number
```

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

Question continues
1.3 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 `rn` means register `n`. (Errors may be detected at 'compile time', or at 'run time', as you prefer.)

1.4 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).

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

1.5 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
```

1.6 A program

Once you are happy with your simulator, **write an RM program** to compute the `n`th triangle number `\sum_{i=0}^{n} i`. Write the program in the file `triangle.r`. This program should **not** contain an initial *registers* line; it should expect to find its input in `r0`, and it should leave the answer in `r0`.

1.7 Submission

For this question, you should tar up your program *executable* (an executable that runs on DICE), and your program *source*, 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 `triangle.r` into a file `rmsim.tar`, and submit this by

```
submit itcs 1 rmsim.tar
```

1.8 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. **No credit:** this is just for fun!
The remaining questions are written questions. If you wish, you may typeset your answers and submit electronically by submit itcs 1 rest.pdf but there is no obligation or expectation to do so. If you hand-write, then submit the paper to the ITO, or submit a clean scan electronically – no mobile phone photos.

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.

2 Reductions for undecidability

In lectures, we considered the Halting Problem $H$, and the Uniform Halting Problem $UH$. Now we consider the Existential Halting problem $EH$: given a machine $M$, is there some input $R$ on which $M$ halts?

(a) Show, by reduction from $H$, that $EH$ is undecidable. [$3$]
(b) Show, by constructing a suitable machine, that $EH$ is semi-decidable. (Hint: interleaving.) [$3$]

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. [$4$]

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.
3 Partial functions

When we defined computable functions, we were talking about standard mathematical functions, which are by definition total.

A partial function \( f : \mathbb{N} \to \mathbb{N} \) is a function \( \mathbb{N} \to \mathbb{N} \cup \{\bot\} \), or equivalently a function that may be undefined on some values (we write \( f(n) = \bot \), or sometimes \( f(n)\uparrow \)).

A partial function \( f \) is computable iff there is a register machine which, given \( n \in R_0 \), halts with \( f(n) \) in \( R_0 \) whenever \( f(n) \neq \bot \).

(a) Let \( \hat{H} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) be the partial function given by

\[
\hat{H}(m, n) = \begin{cases} 
0 & \text{if } m \text{ is not the code of any RM program } P \\
1 & \text{if } m = [\![P]\!] \text{ for some } P, \text{ and } P \text{ halts on input } n \\
\bot & \text{otherwise}
\end{cases}
\]

Show that \( \hat{H} \) is computable. Deduce that it is undecidable whether a computable partial function is total.

(b) It is (easily) decidable whether a number \( n \) is the code \( [\![P]\!] \) of a machine. Therefore we can computably list machines in some order \( P_0, P_1, \ldots \), where \( [\![P_0]\!] \) is the first valid code of a program, and so on.

Consider the partial function \( d : \mathbb{N} \to \mathbb{N} \) given by

\[
d(n) = \begin{cases} 
P_n(n) + 1 & \text{if } P_n \text{ returns a result on input } n \\
\bot & \text{otherwise}
\end{cases}
\]

Is \( d \) computable? Justify your answer.

(c) Now suppose that \( f \) is a total function which agrees with \( d \) (i.e. \( f(n) = d(n) \)) wherever \( d \) is defined. Show that \( f \) is not computable.

4 P, NP

(a) (Revision of big-O notation.) Which of the following are true:

\[
n^2 = O(n \lg n); \ n \lg n = O(n^2); \ 3^n = O(2^n); \ 3^n = 2^{O(n)}.
\]

(b) Suppose that \( X, Y \) are both decision problems over the same domain, and both in \( P \). Show that \( X \cup Y, X \cap Y \) and \( \neg X \) are also in \( P \).

(c) Suppose that \( L_1, L_2 \) are languages of strings over some finite alphabet, whose decision problems are in \( NP \). Let \( L = L_1 L_2 = \{ x_1 x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2 \} \). Show that \( L \in \text{NP} \).