02240 Computability and Semantics Thomas Bolander
Informatics and Mathematical Modelling

Mandatory Assignment 3

This mandatory assignment must be solved in groups of 1, 2 or 3 persons. The solution must be handed in no later than Wednesday, April 20.

Initially you should download the file

http://www.imm.dtu.dk/courses/02240/assignment3.sml

In this file you will find fragments of the SML program that the exercises below ask you to complete.

The purpose of the assignment is to build a compiler that takes an arbitrary **While** program and translates it into a corresponding Turing machine. More precisely, you will declare an SML function `WhileComp` that takes a **While** statement S and returns a Turing machine M which executes the statement S. To simplify the construction, the compiler is split into two parts. The first part compiles **While** statements into *code*, that is, sequences of instructions for the abstract machine. The second part compiles code into Turing machines. The first part will be realised by an SML function `CS` and the second part by an SML function `AMcomp`. The situation is illustrated below:

$$\begin{tikzcd} \textbf{While} \arrow[rr, "\texttt{WhileComp}"] \arrow[dr, "\texttt{CS}"'] & & \text{Turing machines} \\ & \textbf{Code} \arrow[ur, "\texttt{AMcomp}"'] & \end{tikzcd}$$

Part I below concerns the declaration of `CS` and Part II concerns `AMcomp`.

Part I

To simplify the construction we choose a slightly reduced instruction set for the abstract machine (**AM**):

$$\begin{array}{rcl} \mathit{inst} & ::= & \text{PUSH-}n \mid \text{ADD} \mid \text{MULT} \mid \text{SUB} \\ & \mid & \text{EQ} \mid \text{FETCH-}x \mid \text{STORE-}x \mid \text{NOOP} \\ & \mid & \text{BRANCH}(c, c) \mid \text{LOOP}(c, c) \end{array}$$

The instruction set is modelled by the following datatype:

```
datatype inst =
    PUSH of NUM | ADD | MULT | SUB
  | EQ | FETCH of VAR | STORE of VAR | NOOP
  | BRANCH of inst list * inst list | LOOP of inst list * inst list
```

Sequences of instructions are represented as values of type `Code`:

```
type Code = inst list
```

We define translation of statements into code slightly different than in the book:

$$
\begin{array}{ll}
\mathcal{CA}[\![n]\!] & = \text{PUSH-}n \\
\mathcal{CA}[\![x]\!] & = \text{FETCH-}x \\
\mathcal{CA}[\![a_1 + a_2]\!] & = \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \text{ADD} \\
\mathcal{CA}[\![a_1 \star a_2]\!] & = \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \text{MULT} \\
\mathcal{CA}[\![a_1 - a_2]\!] & = \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \text{SUB} \\
\mathcal{CB}[\![\texttt{true}]\!] & = \text{PUSH-}1 \\
\mathcal{CB}[\![\texttt{false}]\!] & = \text{PUSH-}0 \\
\mathcal{CB}[\![a_1 = a_2]\!] & = \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \text{EQ} \\
\mathcal{CB}[\![a_1 \leq a_2]\!] & = \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \text{SUB} : \text{PUSH-}0 : \text{EQ} \\
\mathcal{CB}[\![\neg b]\!] & = \mathcal{CB}[\![b]\!] : \text{PUSH-}0 : \text{EQ} \\
\mathcal{CB}[\![b_1 \wedge b_2]\!] & = \mathcal{CB}[\![b_1]\!] : \mathcal{CB}[\![b_2]\!] : \text{MULT} \\
\mathcal{CS}[\![x := a]\!] & = \mathcal{CA}[\![a]\!] : \text{STORE-}x \\
\mathcal{CS}[\![\texttt{skip}]\!] & = \text{NOOP} \\
\mathcal{CS}[\![S_1; S_2]\!] & = \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = \mathcal{CB}[\![b]\!] : \text{BRANCH}(\mathcal{CS}[\![S_1]\!], \mathcal{CS}[\![S_2]\!]) \\
\mathcal{CS}[\![\texttt{while } b \texttt{ do } S]\!] & = \text{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!])
\end{array}
$$

Note that the boolean expressions `true` and `false` are being translated into PUSH-1 and PUSH-0, respectively. This means that in the abstract machine we use the two integers 1 and 0 to represent the booleans TRUE and FALSE.

a) Declare an SML function

```
CA: AEXP -> Code
```

that translates arithmetic expressions of **While** into code for the **AM**.

b) Declare an SML function

```
CB: BEXP -> Code
```

that translates boolean expressions of **While** into code for the **AM**.

c) Declare a function

```
CS: STM -> code
```

that translates **While** statements into code for the **AM**.

d) Use the function `CS` to translate the following program into code for the **AM**.

```
x1:=1; while ¬(x0=1) do (x1:=x1*x0; x0:=x0-1)
```

Describe and illustrate in detail how the translated program will be executed on the **AM** when started in a state s with $s(\texttt{x0}) = 4$.

e) The **While** statement of question d) computes the factorial function of positive integers. Is it possible to construct a **While** statement computing the same function but using only *a single* variable `x0`? If yes, then provide such a **While** statement. If no, then argue why not. Is it possible to construct code for the abstract machine that computes the same function and uses only a single variable `x0`? If yes, then provide such code. If no, then argue why not.

State	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	$-$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$-$
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	$-$	$-$	$-$

Figure 1: A Turing machine that computes the proper subtraction function.

```
[((q 0,O),(q 1,B,R)),((q 0,I),(q 5,B,R)),
 ((q 1,O),(q 1,O,R)),((q 1,I),(q 2,I,R)),
 ((q 2,O),(q 3,I,L)),((q 2,I),(q 2,I,R)),((q 2,B),(q 4,B,L)),
 ((q 3,O),(q 3,O,L)),((q 3,I),(q 3,I,L)),((q 3,B),(q 0,B,R)),
 ((q 4,O),(q 4,O,L)),((q 4,I),(q 4,B,L)),((q 4,B),(q 6,O,R)),
 ((q 5,O),(q 5,B,R)),((q 5,I),(q 5,B,R)),((q 5,B),(q 6,B,R))]
```

Figure 2: The SML representation of the transition table in Figure 1.

Part II

The purpose of this part is to construct the compiler from code to Turing machines. For simplicity we will only be considering Turing machines that adhere to the following conventions:

- The states are named $q_0, q_1, q_2, \ldots, q_n$ for some n. The state q_0 is always taken to be the start state, and the state with the highest index, q_n, is the final state.
- The set of tape symbols is $\{1, 0, X, B\}$ where B is the blank symbol and the rest are input symbols.

States, tape symbols and directions are modelled in SML by the following datatypes:

```
datatype State = q of int
datatype TapeSymbol = I | O | X | B
datatype Direction = L | R
```

Turing machines can be represented by their transition relations. In SML, we represent these transition relations as objects of type `TransRel`:

```
type TransRel = ((State * TapeSymbol) * (State * TapeSymbol * Direction)) list
```

Example. In Example 8.4 of *Introduction to Automata Theory, Languages, and Computation*, a Turing machine for computing the proper subtraction function is given by the transition table presented in Figure 1. In SML we would represent this table by the value of type `TransRel` given in Figure 2.

In the databar you have access to a Turing machine simulator called xTuring. To run xTuring you should type the following command in a terminal window from the directory containing `assignment3.sml`:

```
~v6474/xturing
```

The simulator can be used to simulate Turing machines represented as SML values of type `TransRel`. This is done by using the function

```
MakeXTfile: TransRel -> string -> unit
```

declared in `assignment3.sml`. Given a Turing machine `M` and a filename `s`, writing

```
MakeXTfile M s
```

will produce a file `s` that can be loaded into xTuring to simulate the Turing machine `M`.

a) Test the Turing machine simulator on the subtraction machine given in Figure 2. You should first declare the list in Figure 2 as an SML value `submachine` and then execute

```
MakeXTfile submachine "submachine.xt"
```

Then you should load the file `submachine.xt` into xTuring. You can then write a string of tape symbols on the tape, move the head to the desired initial position and click 'Run'. You should test the Turing machine on the following initial ID's:

$$q_0 0010$$
$$q_0 000100$$
$$q_0 100$$
$$q_0 001$$

Explain what is calculated in each case (cf. Example 8.4).

b) Give the graphical representation of a Turing machine M_{add} that converts an ID of the form

$$\alpha 1 0^m 1 0^n q_0 B$$

to

$$\alpha 1 0^{m+n} q_f B$$

where α is any string of tape symbols, n and m are non-negative integers, and q_f is the final state of the machine. Represent the machine as an SML value `ADDmachine` of type `TransRel` and test the machine using xTuring.

The program `assignment3.sml` provides a function for composing Turing machines:

```
compose:  TransRel list -> TransRel
```

Suppose $M_1, M_2, \ldots, M_n$ are Turing machines represented as elements of type `TransRel`. Then

$$\texttt{compose } [M_1, M_2, \ldots, M_n]$$

will be the Turing machine that operates as follows: It starts in the initial state of M_1; it then operates as M_1 would operate until M_1 reaches its final state; then it initiates M_2 and operates as M_2 would operate until M_2 reaches its final state; then it initiates M_3 and operates as M_3 would operate until M_3 reaches its final state—and so on, until finally M_n reaches its final state. You will probably find the function `compose` helpful when solving the exercises below.

c) Modify the subtraction Turing machine given in Figure 1 such that it converts an ID of the form

$$\alpha 1 0^m 1 0^n q_0 B$$

to

$$\alpha 1 0^{m \dot{-} n} q_f B$$

where α is any string of tape symbols, n and m are non-negative integers, and q_f is the final state of the machine. The function $\dot{-}$ is defined in Example 8.4 of *Introduction to Automata Theory, Languages, and Computation*. Represent the modified machine as an SML value `SUBmachine` of type `TransRel` and test the machine using xTuring.

d) Example 8.8 of *Introduction to Automata Theory, Languages, and Computation* gives a Turing machine for multiplying two positive integers. The `TransRel` value corresponding to this machine is declared as `multmachine0` in `assignment3.sml`. The book claims that the TM will start with $0^m 1 0^n$ on its tape and end with 0^{mn} on the tape (where m and n are positive integers). Use the xTuring simulator to see that this is *not* the case, and explain what the problem is. Declare a modified Turing machine `multmachine1` that correctly behaves as claimed in the book. Extend the Turing machine `multmachine1` to a Turing machine `multmachine2` such that it converts $0^m 1 0^n$ to 0^{mn} even when m and n are allowed to be 0. Test `multmachine2` using xTuring.

e) Modify the multiplication machine `multmachine2` such that it converts an ID of the form

$$\alpha 1 0^m 1 0^n q_0 B$$

to

$$\alpha 1 0^{mn} q_f B$$

where α is any string of tape symbols, n and m are non-negative integers, and q_f is the final state of the machine. Represent the modified machine as an SML value `MULTmachine` of type `TransRel` and test the machine using xTuring.

As mentioned, the goal of this part of the assignment is to construct a compiler from code to Turing machines. This involves finding a way to represent the *configurations* of the abstract machine by corresponding *ID*'s of Turing machines. Suppose we are given a configuration of the abstract machine in which the stack is

$$n_0 : n_1 : n_2 : n_3 : \cdots : n_k$$

and the state is

$$s = [x_0 \to m_0, x_1 \to m_1, x_2 \to m_2, \ldots, x_l \to m_l].$$

Then this configuration will be represented by a Turing machine ID in which the non-blank part of the tape consists of:

$$\underset{\text{marker}\,\uparrow}{X}\;\overbrace{\underbrace{10^{m_l}}_{\text{representing } s(x_l)}\;\cdots\;\underbrace{10^{m_1}}_{\text{representing } s(x_1)}\;\underbrace{10^{m_0}}_{\text{representing } s(x_0)}}^{\text{representation of the state } s}\;\underset{\text{marker}\,\uparrow}{X}\;\overbrace{\underbrace{10^{n_k}}_{\text{representing } n_k}\;\cdots\;\underbrace{10^{n_1}}_{\text{representing } n_1}\;\underbrace{10^{n_0}}_{\text{representing } n_0}}^{\text{representation of the stack}}$$

Most of the compiler is already included in `assignment3.sml`. The file contains the following declarations:

```
fun InstComp (PUSH n) = writemachine (N(n))
  | InstComp EQ = EQmachine
  | InstComp (FETCH x) = FETCHmachine (getIndex x)
  | InstComp (STORE x) = STOREmachine (getIndex x)
  | InstComp NOOP = NOOPmachine
  | InstComp (BRANCH (am1,am2)) = branchmachine (AMcomp am1) (AMcomp am2)
  | InstComp (LOOP (am1,am2)) = loopmachine (AMcomp am1) (AMcomp am2)
and AMcomp instlist = compose (map InstComp instlist);
```

The function `InstComp` compiles individual instructions for the **AM** into corresponding Turing machines. The function `AMcomp` compiles lists of instructions (code) into Turing machines by composing the Turing machines produced by `InstComp`.

f) The function `InstComp` has not been defined for the following instructions: ADD, MULT, and SUB. Use your Turing machines constructed in questions b), c) and e) to extend `InstComp` to be defined on these instructions as well.

   You now have a compiler `AMcomp` compiling arbitrary lists of instructions into corresponding Turing machines. The entire compiler from **While** statements into Turing machines can then simply be declared by

   ```
   fun WhileComp stm = AMcomp (CS stm)
   ```

   (cf. the figure on p. 1). Test the compiler `WhileComp` on the following **While** statement for computing the factorial of 4:

   ```
   x0:=4; x1:=1; while ¬(x0=1) do (x1:=x1*x0; x0:=x0-1)
   ```

   The resulting Turing machine should be loaded into xTuring and simulated. If your compiler works correctly, the execution will terminate with the following tape content:

$$\underset{\substack{\uparrow \\ \text{marker}}}{X}\ \overbrace{\underbrace{10^{24}}_{\text{representing } s(x_1)\,=\,4!}\ \underbrace{10}_{\text{representing } s(x_0)\,=\,1}}^{\text{terminal state } s}\ \underset{\substack{\uparrow \\ \text{marker}}}{X}\ \overbrace{BBBBBB}^{\text{empty stack}}$$