CO240 - Models of Computation Tutorial Sheets

Tutorial 1 - Expressions

1. Consider the **big-step** operational semantics for the language *SimpleExp* given in the lectures. Find a number n such that

$$(4+1)+(2+2) \Downarrow n$$

Give the full derivation tree.

$$\text{(B-ADD)}\ \dfrac{\text{(B-ADD)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{4 \Downarrow 4} \quad \text{(B-NUM)}\ \dfrac{}{1 \Downarrow 1}}{(4+1) \Downarrow 5} \quad \text{(B-ADD)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{2 \Downarrow 2} \quad \text{(B-NUM)}\ \dfrac{}{2 \Downarrow 2}}{(2+2) \Downarrow 2}}{(4+1)+(2+2) \Downarrow 9}$$

2. The big-step operation semantics for *SimpleExp* was only given for addition. Extend it to include *multiplication*. Give a proof that $((3+2)\times(1+4)) \Downarrow 25$

To do this, we need to add an additional rule as follows;

$$\text{(B-MUL)}\ \dfrac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 \times E_2 \Downarrow n_3}\ n_3 = n_1 \times n_2$$

Hence we can do the following;

$$\text{(B-MUL)}\ \dfrac{\text{(B-ADD)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{3 \Downarrow 3} \quad \text{(B-NUM)}\ \dfrac{}{2 \Downarrow 2}}{(3+2) \Downarrow 5} \quad \text{(B-ADD)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{1 \Downarrow 1} \quad \text{(B-NUM)}\ \dfrac{}{4 \Downarrow 4}}{(1+4) \Downarrow 5}}{((3+2)\times(1+4)) \Downarrow 25}$$

3. Extend the **big-step** semantics further to include *subtraction*. Remember that the numbers in the syntax of the language are $0, 1, 2, \ldots$ (no negative numbers).

How is an expression such as $(3-7)$ handled in your semantics? Have you made any arbitrary decisions about this? If so, what other options were available?

Note that this question has multiple valid options; we can either introduce a `NaN` concept, representing an "invalid" operation, which has to be propagated in all rules, or we could have it be some value. The latter can lead to ambiguity, because if we had $(3-7) \Downarrow 0$, and also $(4-7) \Downarrow 0$, we may unexpected results.

4. Recall the **small-step** operational semantics of *SimpleExp*.

   (a) Give the full derivation of the first step of evaluation of $((1+2)+(4+3))$ - give the derivation tree of the step (for some expression E);

$$((1+2)+(4+3)) \rightarrow E$$

For the first step, we have the following;

$$\text{(S-LEFT)}\ \dfrac{\text{(S-ADD)}\ \dfrac{}{(1+2) \rightarrow 3}}{((1+2)+(4+3)) \rightarrow (3+(4+3))}$$

   (b) Write down all the steps of evaluation needed to reduce the above expression to 10. Give the full derivation for each of these steps.

Note that the **evaluation path** is;

$$((1+2)+(4+3)) \rightarrow (3+(4+3)) \rightarrow (3+7) \rightarrow 10$$

The derivation tree for each step is as follows;

$$\dfrac{\text{(S-ADD)}\ \dfrac{}{(4+3) \to 7}}{(3+(4+3)) \to (3+7)}\ \text{(S-RIGHT)}$$

Followed by;

$$\text{(S-ADD)}\ \dfrac{}{(3+7) \to 10}$$

5. Here is the abstract syntax for a simple language *Bool* of boolean expressions:

$$B \in \mathit{Bool} ::= \texttt{true} \mid \texttt{false} \mid B\&B \mid \neg B \mid \texttt{if}\ B\ \texttt{then}\ B\ \texttt{else}\ B$$

Intuitively, every expression evaluates to either `true` or `false`.

(a) Give a **small-step** operational semantics for *Bool*.

$$\dfrac{B_1 \to B_1'}{B_1\&B_2 \to B_1'\&B_2}$$

$$\dfrac{B_2 \to B_2'}{\texttt{true}\&B_2 \to \texttt{true}\&B_2'}$$

$$\dfrac{B_2 \to B_2'}{\texttt{false}\&B_2 \to \texttt{false}\&B_2'}$$

$$\dfrac{}{\texttt{true}\&\texttt{true} \to \texttt{true}}$$

$$\dfrac{}{\texttt{true}\&\texttt{false} \to \texttt{false}}$$

$$\dfrac{}{\texttt{false}\&\texttt{true} \to \texttt{false}}$$

$$\dfrac{}{\texttt{false}\&\texttt{false} \to \texttt{false}}$$

$$\dfrac{B \to B'}{\neg B \to \neg B'}$$

$$\dfrac{}{\neg\texttt{true} \to \texttt{false}}$$

$$\dfrac{}{\neg\texttt{false} \to \texttt{true}}$$

$$\dfrac{B_1 \to B_1'}{\texttt{if}\ B_1\ \texttt{then}\ B_2\ \texttt{else}\ B_3}$$

$$\dfrac{}{\texttt{if true then}\ B_2\ \texttt{else}\ B_3 \to B_2}$$

$$\dfrac{}{\texttt{if false then}\ B_2\ \texttt{else}\ B_3 \to B_3}$$

Note that these are all evaluated right-to-left.

(b) Write down all the steps of evaluation needed to reduce the following expression to a result:

$$\begin{aligned}
&\neg(\texttt{if (false\&true) then (if true then (false\&true) else false) else }\neg\texttt{true})\\
\to\ &\neg(\texttt{if false then (if true then (false\&true) else false) else }\neg\texttt{true})\\
\to\ &\neg(\neg\texttt{true})\\
\to\ &\neg\texttt{false}\\
\to\ &\texttt{true}
\end{aligned}$$

6. The syntax of *SimpleExp* is extended with a new operator ?, as follows;

$$E \in \mathit{SimpleExp} ::= \ldots \mid (E?E)$$

This operator allows the implementation to choose to give the result of E_1, or E_2, when given $E_1?E_2$.

(a) Extend the **big-step** operational semantics with rules for ? that capture this meaning.

$$\text{(B-CHOICE-1)}\ \frac{E_1 \Downarrow n_1}{E_1?E_2 \Downarrow n_1} \qquad\qquad \text{(B-CHOICE-2)}\ \frac{E_2 \Downarrow n_2}{E_1?E_2 \Downarrow n_2}$$

(b) For what values of n does $(0?1) + (2?3) \Downarrow n$?

$$\text{(B-ADD)}\ \dfrac{\text{(B-CHOICE-1)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{0 \Downarrow 0}}{(0?1) \Downarrow 0} \quad \text{(B-CHOICE-1)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{2 \Downarrow 2}}{(2?3) \Downarrow 2}}{(0?1) + (2?3) \Downarrow 2}$$

$$\text{(B-ADD)}\ \dfrac{\text{(B-CHOICE-1)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{0 \Downarrow 0}}{(0?1) \Downarrow 0} \quad \text{(B-CHOICE-2)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{3 \Downarrow 3}}{(2?3) \Downarrow 3}}{(0?1) + (2?3) \Downarrow 3}$$

$$\text{(B-ADD)}\ \dfrac{\text{(B-CHOICE-2)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{1 \Downarrow 1}}{(0?1) \Downarrow 1} \quad \text{(B-CHOICE-1)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{2 \Downarrow 2}}{(2?3) \Downarrow 2}}{(0?1) + (2?3) \Downarrow 3}$$

$$\text{(B-ADD)}\ \dfrac{\text{(B-CHOICE-2)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{1 \Downarrow 1}}{(0?1) \Downarrow 1} \quad \text{(B-CHOICE-2)}\ \dfrac{\text{(B-NUM)}\ \dfrac{}{3 \Downarrow 3}}{(2?3) \Downarrow 3}}{(0?1) + (2?3) \Downarrow 4}$$

(c) Is the semantics deterministic? Is it total?

It is not deterministic as we have $0?1 \Downarrow 0$, as well as $0?1 \Downarrow 1$ - but $0 \neq 1$. It is total as it is applies to every expression (for something to be total, we need some number n for every expression E such that $E \Downarrow n$).

7. (a) Extend the **small-step** semantics for *SimpleExp* to handle the ? operator by adding appropriate derivation rules for $\rightarrow$.

$$\text{(S-CHOICE-1)}\ \frac{}{E_1?E_2 \rightarrow E_1} \qquad\qquad \text{(S-CHOICE-2)}\ \frac{}{E_1?E_2 \rightarrow E_2}$$

(b) Give all possible derivations of the first step of evaluation of $(0?1) + (2?3)$.

$$\text{(S-LEFT)}\ \dfrac{\text{(S-CHOICE-1)}\ \dfrac{}{0?1 \rightarrow 0}}{(0?1) + (2?3) \rightarrow 0 + (2?3)} \qquad \text{(S-LEFT)}\ \dfrac{\text{(S-CHOICE-2)}\ \dfrac{}{0?1 \rightarrow 1}}{(0?1) + (2?3) \rightarrow 1 + (2?3)}$$

(c) Give all of the possible evaluation paths for $(0?1) + (2?3)$.

$$\begin{gathered}(0?1) + (2?3) \rightarrow 0 + (2?3) \rightarrow 0 + 2 \rightarrow 2\\ (0?1) + (2?3) \rightarrow 0 + (2?3) \rightarrow 0 + 3 \rightarrow 3\\ (0?1) + (2?3) \rightarrow 1 + (2?3) \rightarrow 1 + 2 \rightarrow 3\\ (0?1) + (2?3) \rightarrow 1 + (2?3) \rightarrow 1 + 3 \rightarrow 4\end{gathered}$$

(d) Is the semantics confluent?

We've shown $(0?1) + (2?3) \rightarrow^* 2$ and also $(0?1) + (2?3) \rightarrow^* 3$. Therefore, for the semantics to be confluent, there must be some E' such that $2 \rightarrow^* E'$ and $3 \rightarrow^* E'$ - however, since they are both in normal forms, they can only evaluate to themselves. $2 \neq 3$, hence it is not confluent.

(e) Is the semantics normalising?

Yes, there are no infinite sequences of expressions, hence any evaluation path will eventually reach a normal form.

8. Suppose that instead of the *SimpleExp* small-step rule (S-RIGHT), we had the following;

$$\text{(S-RIGHT}'\text{)}\ \frac{E_2 \rightarrow E_2'}{(E_1 + E_2) \rightarrow (E_1 + E_2')}$$

(a) Given an evaluation path using the S-RIGHT rule, is it also an evaluation path using the S-RIGHT′ rule?

Yes, as the original rule constrained E_1 to be in a normal form, but the new rule doesn't. This means that the new rule covers all the cases of the original rule.

(b) Find an expression that has an evaluation path using the S-RIGHT′ rule that it did not have with the S-RIGHT rule.

$$(0+1)+(2+3) \rightarrow (0+1)+5 \rightarrow 1+5 \rightarrow 6$$

(c) Is $\rightarrow$ deterministic?

No, starting with $(0+1)+(2+3)$, we can go to either $1+(2+3)$ S-LEFT, or $(0+1)+5$ with S-RIGHT′ - however the two expressions are not equal.

(d) Is $\rightarrow$ confluent?

Yes, the rule allows for different evaluation order, but doesn't change the result of the evaluation.

Tutorial 2 - State

1. Consider the small-step operation semantics of the language *While*. Write down all of the evaluation steps of the program $(z := x; x := y); y := z$, with the initial state $s = (x \mapsto 5, y \mapsto 7)$. Give the full derivation tree for the first step in this evaluation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_e \langle 5, (x \mapsto 5, y \mapsto 7)\rangle}\text{(W-EXP.VAR)}}{\langle z := x, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_c \langle z := 5, (x \mapsto 5, y \mapsto 7)\rangle}\text{(W-ASS.EXP)}}{\langle z := x; x := y, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_c \langle z := 5; x := y, (x \mapsto 5, y \mapsto 7)\rangle}\text{(W-SEQ.LEFT)}}{\langle (z := x; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \rightarrow_c \langle (z := 5; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle}\text{(W-SEQ.LEFT)}$$

All of the steps are as follows;

$$\begin{aligned}
&\langle (z := x; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \\
\rightarrow_c\ &\langle (z := 5; x := y); y := z, (x \mapsto 5, y \mapsto 7)\rangle \\
\rightarrow_c\ &\langle (\texttt{skip}; x := y); y := z, (x \mapsto 5, y \mapsto 7, z \mapsto 5)\rangle \\
\rightarrow_c\ &\langle x := y; y := z, (x \mapsto 5, y \mapsto 7, z \mapsto 5)\rangle \\
\rightarrow_c\ &\langle x := 7; y := z, (x \mapsto 5, y \mapsto 7, z \mapsto 5)\rangle \\
\rightarrow_c\ &\langle \texttt{skip}; y := z, (x \mapsto 7, y \mapsto 7, z \mapsto 5)\rangle \\
\rightarrow_c\ &\langle y := z, (x \mapsto 7, y \mapsto 7, z \mapsto 5)\rangle \\
\rightarrow_c\ &\langle y := 5, (x \mapsto 7, y \mapsto 7, z \mapsto 5)\rangle \\
\rightarrow_c\ &\langle \texttt{skip}, (x \mapsto 7, y \mapsto 5, z \mapsto 5)\rangle
\end{aligned}$$

2. Consider the small-step operational semantics of the language *While*. Write down all of the evaluation steps of the program (given the initial state $s = (x \mapsto 1)$)

$$(\text{let } W =)\ \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2$$

Give full derivation trees for the first four steps.

$$\dfrac{}{\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2, (x \mapsto 1)\rangle \rightarrow_c \langle \texttt{if}\ x < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle}1$$

$$
{\scriptstyle 2}\ \dfrac{{\scriptstyle (\text{W-BEXP.LEFT})}\ \dfrac{{\scriptstyle (\text{W-EXP.VAR})}\ \dfrac{}{\langle x, (x \mapsto 1)\rangle \to_e \langle 1, (x \mapsto 1)\rangle}}{\langle x < 4, (x \mapsto 1)\rangle \to_b \langle 1 < 4, (x \mapsto 1)\rangle}}{\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2, (x \mapsto 1)\rangle \to_c \langle \texttt{if}\ 1 < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle}
$$

$$
{\scriptstyle 2}\ \dfrac{{\scriptstyle (\text{W-BEXP.LT})}\ \dfrac{}{\langle 1 < 4, (x \mapsto 1)\rangle \to_b \langle \texttt{true}, (x \mapsto 1)\rangle}}{\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2, (x \mapsto 1)\rangle \to_c \langle \texttt{if true then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle}
$$

$$
{\scriptstyle (\text{W-COND.TRUE})}\ \dfrac{}{\langle \texttt{if true then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle \to_c \langle x := x + 2; W, (x \mapsto 1)\rangle}
$$

Note that rule 1 is (W-WHILE), and rule 2 is (W-COND.BEXP). The full evaluation path is as follows;

$$
\begin{aligned}
&\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2, (x \mapsto 1)\rangle \\
\to_c\ &\langle \texttt{if}\ x < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle \\
\to_c\ &\langle \texttt{if}\ 1 < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle \\
\to_c\ &\langle \texttt{if true then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 1)\rangle \\
\to_c\ &\langle x := x + 2; W, (x \mapsto 1)\rangle \\
\to_c\ &\langle x := 1 + 2; W, (x \mapsto 1)\rangle \\
\to_c\ &\langle x := 3; W, (x \mapsto 1)\rangle \\
\to_c\ &\langle \texttt{skip}; W, (x \mapsto 3)\rangle \\
\to_c\ &\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2, (x \mapsto 3)\rangle \\
\to_c\ &\langle \texttt{if}\ x < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 3)\rangle \\
\to_c\ &\langle \texttt{if}\ 3 < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 3)\rangle \\
\to_c\ &\langle \texttt{if true then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 3)\rangle \\
\to_c\ &\langle x := x + 2; W, (x \mapsto 3)\rangle \\
\to_c\ &\langle x := 3 + 2; W, (x \mapsto 3)\rangle \\
\to_c\ &\langle x := 5; W, (x \mapsto 3)\rangle \\
\to_c\ &\langle \texttt{skip}; W, (x \mapsto 5)\rangle \\
\to_c\ &\langle \texttt{while}\ x < 4\ \texttt{do}\ x := x + 2, (x \mapsto 5)\rangle \\
\to_c\ &\langle \texttt{if}\ x < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 5)\rangle \\
\to_c\ &\langle \texttt{if}\ 5 < 4\ \texttt{then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 5)\rangle \\
\to_c\ &\langle \texttt{if false then}\ (x := x + 2; W)\ \texttt{else skip}, (x \mapsto 5)\rangle \\
\to_c\ &\langle \texttt{skip}, (x \mapsto 5)\rangle
\end{aligned}
$$

3. Consider adding the increment expression $x++$ to the language *While*. The expression returns the value of the variable (only applied to variables) x and then updates the value of x to be one greater than the old value; its semantics is given by the following rule:

$$
{\scriptstyle (\text{W-EXP.PP})}\ \dfrac{}{\langle x++, s\rangle \to_e \langle n, s[x \mapsto n']\rangle}\ {\scriptstyle s(x) = n,\, n' = n + 1}
$$

   (a) Give the full execution path for the program $x := (x++) + (x++)$ from the initial state $(x \mapsto 2)$.

$$
\begin{aligned}
&\langle x := (x++) + (x++), (x \mapsto 2)\rangle \\
\to_c\ &\langle x := 2 + (x++), (x \mapsto 3)\rangle \\
\to_c\ &\langle x := 2 + 3, (x \mapsto 4)\rangle \\
\to_c\ &\langle x := 5, (x \mapsto 4)\rangle \\
\to_c\ &\langle \texttt{skip}, (x \mapsto 5)\rangle
\end{aligned}
$$

(b) Given an operational semantics rule for $++x$, which increments x and then returns the result.

$$\text{(W-EXP.PP)}\ \frac{}{\langle ++x, s\rangle \to_e \langle n', s[x \mapsto n']\rangle}\ {\scriptstyle s(x) = n, n' = n+1}$$

4. Consider what happens if we add a 'side-effecting expression' of the form

$$\texttt{do}\ C\ \texttt{return}\ E$$

This runs first runs the command C, and returns the value of E.

$$\frac{\langle C, s\rangle \to_c \langle C', s'\rangle}{\langle \texttt{do}\ C\ \texttt{return}\ E, s\rangle \to_e \langle \texttt{do}\ C'\ \texttt{return}\ E, s'\rangle} \qquad \frac{}{\langle \texttt{do skip return}\ E, s\rangle \to_e \langle E, s\rangle}$$

5. Consider the *While* language extend with parallel composition of commands: $C \mid\mid C$. The semantics of parallel composition is given by interleaving the execution steps of the two composed commands in an arbitrary fashion. This is expressed formally as;

$$\frac{\langle C_1, s\rangle \to_c \langle C_1', s'\rangle}{\langle C_1 \mid\mid C_2, s\rangle \to_c \langle C_1' \mid\mid C_2, s'\rangle} \quad \frac{\langle C_2, s\rangle \to_c \langle C_2', s'\rangle}{\langle C_1 \mid\mid C_2, s\rangle \to_c \langle C_1 \mid\mid C_2', s'\rangle} \quad \frac{}{\langle \texttt{skip} \mid\mid \texttt{skip}, s\rangle \to_c \langle \texttt{skip}, s\rangle}$$

(a) Consider the command $(x := 1) \mid\mid (x := 2; x := (x+2))$, run with initial state $s = (x \mapsto 0)$. How many possible final values for x does this command have?

There are 3 possible values; 1, 3, or 4.

(b) How many different evaluation paths exist for obtaining the final value 4?

3 paths. I really can't be bothered to type out all of the steps. The point is the operation $x := x + 2$ is not atomic; even if we have obtained $x := 4$, we can execute $x := 1$, and then still obtain a state with $x \mapsto 4$, if the former is executed at the end.

(c) A useful operation in concurrency is atomic compare-and-swap. This operation is added to the *While* language in the form of a new boolean expression $\texttt{CAS}(x, E, E)$. To execute the operation $\texttt{CAS}(x, E_1, E_2)$, first E_1 and then E_2 are evaluated to numbers n_1 and n_2 in the usual way. Then, **in a single step**, the operation compares the value of variable x with n_1; if the values are equal, it updates the value of x to be number n_2 and returns `true`, otherwise, it simply returns `false`. Extend the operational semantics with rules for `CAS` that implement this behaviour.

$$\frac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle \texttt{CAS}(x, E_1, E_2), s\rangle \to_b \langle \texttt{CAS}(x, E_1', E_2), s'\rangle}$$

$$\frac{\langle E_2, s\rangle \to_e \langle E_2', s'\rangle}{\langle \texttt{CAS}(x, n_1, E_2), s\rangle \to_b \langle \texttt{CAS}(x, n_1, E_2'), s'\rangle}$$

$$\frac{}{\langle \texttt{CAS}(x, n_1, n_2), s\rangle \to_b \langle \texttt{true}, s[x \mapsto n_2]\rangle}\ {\scriptstyle s(x) = n_1}$$

$$\frac{}{\langle \texttt{CAS}(x, n_1, n_2), s\rangle \to_b \langle \texttt{false}, s\rangle}\ {\scriptstyle s(x) \neq n_1}$$

6. Suppose that $\langle C_1; C_2, s\rangle \to_c^* \langle C_2, s'\rangle$. Show that it is not necessarily the case that $\langle C_1, s\rangle \to_c^* \langle \texttt{skip}, s'\rangle$.

Let there be a state $s'' \neq s'$, where $\langle C_1, s\rangle \to_c^* \langle \texttt{skip}, s''\rangle$. For $\langle C_1; C_2, s\rangle \to_c^* \langle C_2, s'\rangle$, we can find C_2 such that $\langle C_2, s''\rangle \to_c^* \langle C_2, s'\rangle$. From here, we see that our foal is to find C_2 as something that evaluates to itself, but in a different state (hence a loop).

$$C_1 = \texttt{skip}$$
$$C_2 = \texttt{while true do}\ x := 1$$
$$s = (x \mapsto 0)$$

Executing this we have;

$$\begin{aligned}
&\langle \texttt{while true do } x := 1, (x \mapsto 0)\rangle \\
\rightarrow_c\ &\langle \texttt{if true then } x := 1; C_2 \texttt{ else skip}, (x \mapsto 0)\rangle \\
\rightarrow_c\ &\langle x := 1; C_2, (x \mapsto 0)\rangle \\
\rightarrow_c\ &\langle \texttt{skip}; C_2, (x \mapsto 1)\rangle \\
\rightarrow_c\ &\langle C_2, (x \mapsto 1)\rangle
\end{aligned}$$

Tutorial 3 - Induction

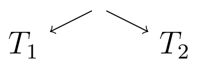
1. s Binary trees are a commonly used data structure. Roughly, a binary tree is either a single leaf node, or a branch node which has two subtrees. The set of binary trees can be defined formally by the following grammar;

$$\texttt{bTree} ::= \texttt{Node} \mid \texttt{Branch}(\texttt{bTree}, \texttt{bTree})$$

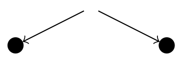
(a) Draw pictures of the following binary trees;

- Node

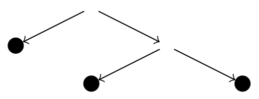
- $\texttt{Branch}(T_1, T_2)$



- $\texttt{Branch}(\texttt{Node}, \texttt{Node})$



- $\texttt{Branch}(\texttt{Node}, \texttt{Branch}(\texttt{Node}, \texttt{Node}))$



(b) d We define th function `leaves` which takes a binary tree as an argument and returns the number of leaf nodes, given by `Node`, in a tree, and similarly `branches`, which counts the number of $\texttt{Branch}(\_, \_)$ nodes in a tree:

$$\begin{aligned}
\texttt{leaves}(\texttt{Node}) &= 1 \\
\texttt{leaves}(\texttt{Branch}(T_1, T_2)) &= \texttt{leaves}(T_1) + \texttt{leaves}(T_2) \\
\texttt{branches}(\texttt{Node}) &= 0 \\
\texttt{branches}(\texttt{Branch}(T_1, T_2)) &= \texttt{branches}(T_1) + \texttt{branches}(T_2) + 1
\end{aligned}$$

Prove by induction on the structure of trees, that for any tree T;

$$\texttt{leaves}(T) = \texttt{branches}(T) + 1$$

This trivially checks out for the base case, as we have

$$\texttt{leaves}(\texttt{Node}) = 1 = 0 + 1 = \texttt{branches}(\texttt{Node}) + 1$$

For the inductive step, let $T = \texttt{Branch}(T_1, T_2)$, and assume that this holds for T_1 and T_2;

$$\begin{aligned}
\texttt{leaves}(T_1) &= \texttt{branches}(T_1) + 1 && \text{inductive hypothesis} \\
\texttt{leaves}(T_2) &= \texttt{branches}(T_2) + 1 && \text{inductive hypothesis} \\
\texttt{leaves}(T) &= \texttt{leaves}(T_1) + \texttt{leaves}(T_2) && \text{by def. of } \texttt{leaves} \\
&= \texttt{branches}(T_1) + 1 + \texttt{branches}(T_2) + 1 && \text{by substitution} \\
&= \texttt{branches}(\texttt{Branch}(T_1, T_2)) + 1 && \text{by def. of } \texttt{branches} \\
&= \texttt{branches}(T) + 1 && \blacksquare
\end{aligned}$$

2. Recall the **big-step** operational semantics for simple expressions E. Prove by structural induction on the structure of expressions that, for every E, there is some number n such that $E \Downarrow n$.

$$E \in \mathit{SimpleExp} ::= n \mid E + E$$

Trivially, for the base case, $n \Downarrow n$. For the case where we have $E = E_1 + E_2$, assume this holds for E_1 and E_2, such that $E_1 \Downarrow n_1$ and $E_2 \Downarrow n_2$. Then $E_1 + E_2 \Downarrow n_3$, by (B-ADD), where $n_3 = n_1 + n_2$.

4. Recall the **small-step** operational semantics for simple expressions. Prove, by induction on the structure of simple expressions, that for every expression E, either $E = n$ for some number n, or $E \to E'$ for some expression E'.

We can first formalise the property as $P(E) \equiv (\exists n.\ E = n) \vee (\exists E'.\ E \to E')$.

Trivially, the base case $P(n)$ (where n is an arbitrary number), holds as n is the number itself. The inductive step has the following inductive hypothesis;

(1) $(\exists n_1.\ E_1 = n_1) \vee (\exists E_1'.\ E_1 \to E_1')$
(2) $(\exists n_2.\ E_2 = n_2) \vee (\exists E_2'.\ E_2 \to E_2')$

For $E = E_1 + E_2$, we can look at the following cases;

- $E_1 = n_1$ and $E_2 = n_2$ $\qquad$ $\text{(S-ADD)}\ \dfrac{}{n_1 + n_2 \to n_3}\ n_3 = n_1 + n_2$
- $E_1 = n_1$ and $E_2 \to E_2'$ $\qquad$ $\text{(S-RIGHT)}\ \dfrac{E_2 \to E_2'}{n_1 + E_2 \to n_1 + E_2'}$
- $E_1 \to E_1'$ $\qquad$ $\text{(S-LEFT)}\ \dfrac{E_1 \to E_1'}{E_1 + E_2 \to E_1' + E_2}$

5. Recall the **small-step** operational semantics for simple expressions.

(a) By induction on the structure of simple expressions, define a function $\texttt{ops} : \mathit{SimpleExp} \to \mathbb{N}$ that gives the number of operators in an expression.

$$\begin{aligned}\texttt{ops}(n) &= 0\\ \texttt{ops}(E_1 + E_2) &= \texttt{ops}(E_1) + \texttt{ops}(E_2) + 1\end{aligned}$$

(b) By induction on the structure of simple expressions, prove that for all simple expressions, E, E', with $E \to E'$, $\texttt{ops}(E) > \texttt{ops}(E')$.

Since the proofs for $+$ and $\times$ are pretty much identical, only the former will be written out. Let us first write this property as $P(E) \equiv \forall E'.\ E \to E' \Rightarrow \texttt{ops}(E) > \texttt{ops}(E')$. This holds trivially for the base case, as there is no E' such that $n \to E'$ for arbitrary n.

For the inductive step, let $E = E_1 + E_2$, hence the inductive hypothesis is;

(1) $P(E_1) \equiv \forall E_1'.\ E_1 \to E_1' \Rightarrow \texttt{ops}(E_1) > \texttt{ops}(E_1')$
(2) $P(E_2) \equiv \forall E_2'.\ E_2 \to E_2' \Rightarrow \texttt{ops}(E_2) > \texttt{ops}(E_2')$

Hence we can use the definition of ops as follows, with three cases corresponding to the rules and axioms;

$$\text{(S-LEFT)}\ \dfrac{E_1 \to E_1'}{E_1 + E_2 \to E_1' + E_2}$$

$$\begin{aligned}\texttt{ops}(E) &= \texttt{ops}(E_1 + E_2) &\\ &= \texttt{ops}(E_1) + \texttt{ops}(E_2) + 1 & \text{by def. of } \texttt{ops}\\ &> \texttt{ops}(E_1') + \texttt{ops}(E_2) + 1 & \text{by inductive hypothesis (1)}\\ &= \texttt{ops}(E_1' + E_2) & \text{by def. of } \texttt{ops}\end{aligned}$$

$$= \texttt{ops}(E')$$

$$\text{(S-RIGHT)}\ \frac{E_2 \to E_2'}{n_1 + E_2 \to n_1 + E_2'} \qquad E_1 = n_1$$

$$\begin{aligned}
\texttt{ops}(E) &= \texttt{ops}(n_1 + E_2) \\
&= \texttt{ops}(n_1) + \texttt{ops}(E_2) + 1 && \text{by def. of } \texttt{ops} \\
&> \texttt{ops}(n_1) + \texttt{ops}(E_2') + 1 && \text{by inductive hypothesis (2)} \\
&= \texttt{ops}(n_1 + E_2') && \text{by def. of } \texttt{ops} \\
&= \texttt{ops}(E')
\end{aligned}$$

$$\text{(S-ADD)}\ \frac{}{n_1 + n_2 \to n_3}\ n_3 = n_1 + n_2 \qquad E_1 = n_1 \text{ and } E_2 = n_2$$

$$\begin{aligned}
\texttt{ops}(E) &= \texttt{ops}(n_1 + n_2) \\
&= \texttt{ops}(n_1) + \texttt{ops}(n_2) + 1 && \text{by def. of } \texttt{ops} \\
&= 1 && \text{by def. of } \texttt{ops} \\
&> 0 \\
&= \texttt{ops}(n_3) && \text{by def. of } \texttt{ops} \\
&= \texttt{ops}(E')
\end{aligned}$$

Hence it follows for all E.

(c) Hence or otherwise, prove that $\to$ is normalising.

As each evaluation causes `ops` to decrease, we know it will eventually terminate as `ops` will reach 0. When it does reach 0, it will be a number, hence it must eventually reach this normal form.

6. For any simple expression E, prove by induction on the structure of expressions that;

$$E \Downarrow n \text{ if and only if } E \to^* n$$

First, let us define one side of the implication as $P(E) \equiv E \Downarrow n \Rightarrow E \to^* n$. For the base case, $P(n)$ (arbitrary n) trivially holds, as we have $E = n$, hence $n \to^0 n$, and $n \Downarrow n$.

For the inductive step, let $E = E_1 + E_2$, and first assume $(E_1 + E_2) \Downarrow n$.

$$\text{(B-ADD)}\ \frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n}\ n = n_1 + n_2$$

The inductive hypothesis is therefore;

(1) $P(E_1) \equiv E_1 \Downarrow n_1 \Rightarrow E_1 \to^* n_1$

(2) $P(E_2) \equiv E_2 \Downarrow n_2 \Rightarrow E_2 \to^* n_2$

By (1), we can write;

$$(E_1 + E_2) \to (E_1' + E_2) \to \cdots \to (n_1 + E_2)$$

Similarly, by using (2), we can write;

$$(n_1 + E_2) \to (n_1 + E_2') \to \cdots \to (n_1 + n_2) \to n$$

Therefore, we have $(E_1 + E_2) \to^* n$, which gives us $E \to^* n$. Hence $(E_1 + E_2) \Downarrow n \Rightarrow E \to^* n$.

On the other hand, we can prove the other direction using previous results. Assume that $E \to^* n$, and by totality of $\Downarrow$, we have $E \Downarrow m$ for some m. By determinacy of *SimpleExp*, we know $E \to^* m$ and $E \to^* n$ only holds when $m = n$, hence $E \Downarrow n$.

Tutorial 4 - Register Machines

1. Consider the register machine given by the following code:

$$
\begin{aligned}
&L_0 : R_1^- \to L_1, L_7\\
&L_1 : R_0^+ \to L_2\\
&L_2 : R_2^- \to L_3, L_5\\
&L_3 : R_3^+ \to L_4\\
&L_4 : R_0^+ \to L_1\\
&L_5 : R_2^+ \to L_6\\
&L_6 : R_3^- \to L_5, L_0\\
&L_7 : \texttt{HALT}
\end{aligned}
$$

   (a) Give the graphical representation of the register machine.

$$\begin{tikzcd}
\texttt{START} \arrow[r] & R_1^- \arrow[r, two heads] \arrow[d] & \texttt{HALT} \\
R_0^+ \arrow[r] & R_0^+ \arrow[d] & R_3^- \arrow[ul] \arrow[d, bend left] \\
R_3^+ \arrow[u] & R_2^- \arrow[l] \arrow[r, two heads] & R_2^+ \arrow[u, bend left]
\end{tikzcd}$$

   (b) Give the computation when the register machine is run from the initial configuration $(0, 0, 2, 0, 0)$.

L	R_0	R_1	R_2	R_3
0	0	2	0	0
1	0	1	0	0
2	1	1	0	0
5	1	1	0	0
6	1	1	1	0
0	1	1	1	0
1	1	0	1	0
2	2	0	1	0
3	2	0	0	0
4	2	0	0	1
1	3	0	0	1
2	4	0	0	1
5	4	0	0	1
6	4	0	1	1
5	4	0	1	0
6	4	0	2	0
0	4	0	2	0
7	4	0	2	0

2. In this question, you will design register machines that implement subtraction.

   (a) Consider the function $f(x_1, x_2)$ defined as;

$$f(x_1, x_2) \triangleq \begin{cases} x_1 - x_2 & \text{if } x_1 \geq x_2 \\ 0 & \text{otherwise} \end{cases}$$

      i. Define a register machine that computes the function f

$$L_0 : R_2^- \to L_1, L_2$$

$L_1 : R_1^- \to L_0, L_4$
$L_2 : R_1^- \to L_3, L_4$
$L_3 : R_0^+ \to L_4$
$L_4 : \texttt{HALT}$

ii. Draw the graph corresponding to the register machine.

$$\begin{tikzcd}
\texttt{START} \arrow[r] & R_2^- \arrow[r, bend left] \arrow[d, two heads] & R_1^- \arrow[l, bend left] \arrow[d, two heads] \\
R_0^+ \arrow[r, bend left] & R_1^- \arrow[l, bend left] \arrow[r, two heads] & \texttt{HALT}
\end{tikzcd}$$

(b) Consider the partial function $g(x_1, x_2)$ defined as;

$$g(x_1, x_2) \triangleq \begin{cases} x_1 - x_2 & \text{if } x_1 \geq x_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

i. How would a register machine implementing $g(x_1, x_2)$ behave when $x_2 > x_1$?

For a register machine to implement $g(x_1, x_2)$, it must halt with $R_0 = y$ when $g(x_1, x_2) = y$. However, since there is no such y, it cannot halt.

ii. By adapting your answer to part (a), define a register machine that computes the partial function g.

Since we want the machine to not terminate when $x_2 > x_1$, L_1 needs to be modified to cause an infinite loop. The easiest way to do this is to change L_1 to be $L_1 : R_1^- \to L_0, L_1$, thus cycling back to itself.

$$\begin{tikzcd}
\texttt{START} \arrow[r] & R_2^- \arrow[r, bend left] \arrow[d, two heads] & R_1^- \arrow[l, bend left] \arrow[loop right, two heads] \\
R_0^+ \arrow[r, bend left] & R_1^- \arrow[l, bend left] \arrow[r, two heads] & \texttt{HALT}
\end{tikzcd}$$

3. Consider the register machine represented by the following graph:

$$\begin{tikzcd}
\texttt{START} \arrow[r] & R_1^- \arrow[r, bend left] \arrow[d, two heads] & R_1^- \arrow[l, bend left] \arrow[d, two heads] \\
 & \texttt{HALT} & R_0^+ \arrow[l]
\end{tikzcd}$$

(a) Give the code of the register machine.

$L_0 : R_1^- \to L_1, L_3$
$L_1 : R_1^- \to L_0, L_2$
$L_2 : R_0^+ \to L_3$
$L_3 : \texttt{HALT}$

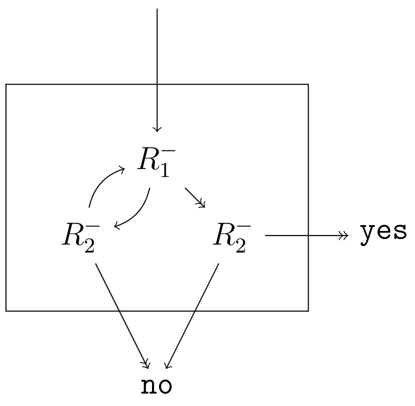
(b) Describe the function of one argument, $f(x)$, that is computed by the register machine.

$$f(x) = \begin{cases} 1 & x \text{ is odd} \\ 0 & x \text{ is even} \end{cases}$$

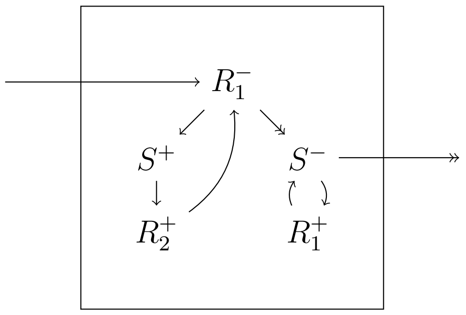
Same as computing the remainder of x divided by 2.

4. In order to construct register machines to perform complex operations, it is useful to build them from smaller components that we'll call gadgets, which perform specific operations.

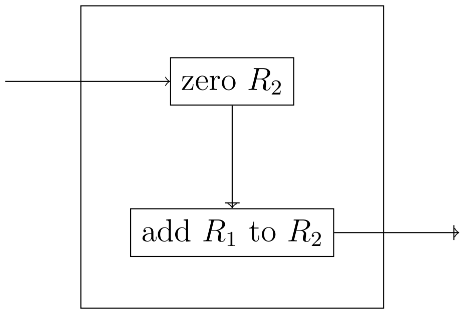
   For example, the following gadget tests whether $R_1 = R_2$;



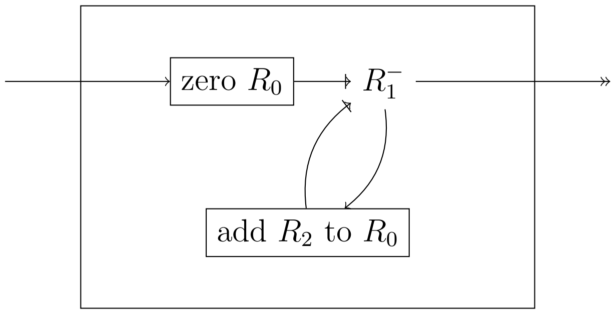
   (a) Define a gadget "add R_1 to R_2", which adds the initial value of R_1 to register R_2, storing the result in R_2 but restoring R_1 to its initial value (use a scratch register initialised to 0, but must also be reset to 0 after exiting the gadget).



   (b) Define a gadget "copy R_1 to R_2", which copies the value of R_1 into register R_2, leaving R_1 with its initial value.



   (c) Define a gadget "multiply R_1 by R_2 to R_0", which multiples R_1 by R_2, and stores the result in R_0 (possibly overwriting the initial values).



   (d) Define a gadget "test $R_1 < R_2$" which determines whether the initial value of R_1 is less than that of R_2 (possibly overwriting the initial values).

$$\begin{tikzcd}
& {} \arrow[d] & & \\
& R_1^- \arrow[dl, bend right] \arrow[dr] & & \\
R_2^- \arrow[ur, bend left] \arrow[dr] & & R_2^- \arrow[r] \arrow[dl] & \texttt{yes} \\
& \texttt{no} & &
\end{tikzcd}$$

(e) Describe the function of one argument $f(x)$ computed by the register machine M defined above.

Not really bothered to draw it; starts with R_2^+, then to copy R_2 to R_3, then copy R_2 to R_4, then multiplies R_3 by R_4 to R_6. copies R_1 to R_5. It then does a test whether $R_5 < R_6$, if it is, then it halts, otherwise it does R_0^+, and goes **back** to R_2^+.

This computes the greatest value $f(x)$ such that $(f(x))^2 \leq x$, hence it computes the floor of the positive square root of x.

$$f(x) = \lfloor \sqrt{x} \rfloor$$

Tutorial 5 - More Register Machines

1. Consider the register machine program P, given by the following code;

$$\begin{aligned}
&L_0 : R_1^- \to L_1, L_6 \\
&L_1 : R_2^- \to L_2, L_4 \\
&L_2 : R_0^+ \to L_3 \\
&L_3 : R_3^+ \to L_1 \\
&L_4 : R_3^- \to L_5, L_0 \\
&L_5 : R_2^+ \to L_4 \\
&L_6 : \texttt{HALT}
\end{aligned}$$

Which computes the function $f(x, y) = x \times y$. The code of P has the form $\ulcorner[\ulcorner B_0 \urcorner, \dots, \ulcorner B_6 \urcorner]\urcorner$, where B_i is the body of L_i. Give the value of $\ulcorner B_i \urcorner$ for each i.

$$\begin{aligned}
\ulcorner B_0 \urcorner &= \langle\langle 2 \cdot 1 + 1, \langle 1, 6 \rangle \rangle\rangle \\
&= \langle\langle 3, 25 \rangle\rangle \\
&= 408 \\
\ulcorner B_1 \urcorner &= \langle\langle 2 \cdot 2 + 1, \langle 2, 4 \rangle \rangle\rangle \\
&= \langle\langle 5, 35 \rangle\rangle \\
&= 2272 \\
\ulcorner B_2 \urcorner &= \langle\langle 2 \cdot 0, 3 \rangle\rangle \\
&= \langle\langle 0, 3 \rangle\rangle \\
&= 7 \\
\ulcorner B_3 \urcorner &= \langle\langle 2 \cdot 3, 1 \rangle\rangle \\
&= \langle\langle 6, 1 \rangle\rangle \\
&= 192 \\
\ulcorner B_4 \urcorner &= \langle\langle 2 \cdot 3 + 1, \langle 5, 0 \rangle \rangle\rangle
\end{aligned}$$

$$
\begin{aligned}
&= \langle\!\langle 7, 31 \rangle\!\rangle \\
&= 8064 \\
\ulcorner B_5 \urcorner &= \langle\!\langle 2 \cdot 2, 4 \rangle\!\rangle \\
&= \langle\!\langle 4, 4 \rangle\!\rangle \\
&= 144 \\
\ulcorner B_6 \urcorner &= 0
\end{aligned}
$$

2. Consider the natural number $2^{216} \cdot 833$.

   (a) What register machine is represented by this number?

   Working backwards from the coding of pairs, we get the following list;

$$[216, 5, 1, 0]$$

   Using these values, we can get the labels as follows;

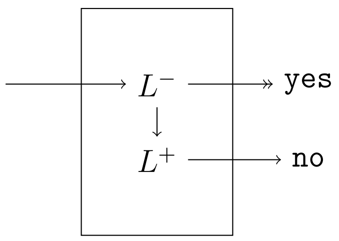
$$
\begin{aligned}
216 &= 2^3(2 \cdot 13 + 1) \\
&= \langle\!\langle 3, 13 \rangle\!\rangle \\
&= \langle\!\langle 2 \cdot 1 + 1, \langle 1, 3 \rangle \rangle\!\rangle \\
&= \ulcorner R_1^- \to L_1, L_3 \urcorner \\
5 &= 2^0(2 \cdot 2 + 1) \\
&= \langle\!\langle 0, 2 \rangle\!\rangle \\
&= \langle\!\langle 2 \cdot 0, 2 \rangle\!\rangle \\
&= \ulcorner R_0^+ \to L_2 \urcorner \\
1 &= 2^0(2 \cdot 0 + 1) \\
&= \langle\!\langle 0, 0 \rangle\!\rangle \\
&= \langle\!\langle 2 \cdot 0, 0 \rangle\!\rangle \\
&= \ulcorner R_0^+ \to L_0 \urcorner \\
0 &= \ulcorner \texttt{HALT} \urcorner
\end{aligned}
$$

   (b) What function of one argument is computed by this register machine?

   It doubles the input, such that $f(x) = 2x$.

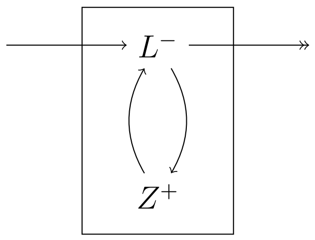
3. (a) Define a gadget which determines whether the initial value of register L is 0, without changing the value;

test $L = 0$

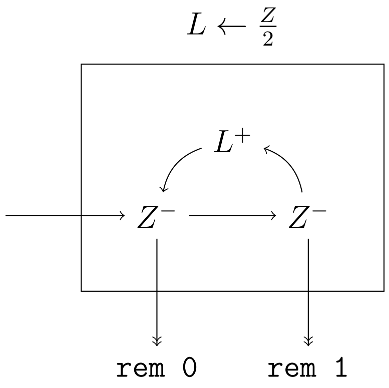


   (b) Define a gadget which, when initially $Z = 0$ and $L = l$, exits with $Z = l$ and $L = 0$;

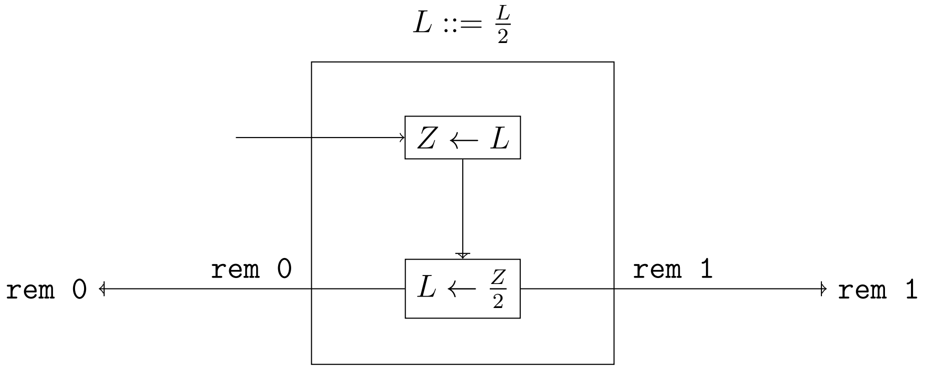
$Z \leftarrow L$



(c) Define a gadget which computes the quotient of Z, such that if we start with $Z = z$ and $L = 0$, we terminate with $Z = 0$ and $L = \lfloor \frac{z}{2} \rfloor$, exiting on the `rem 0` branch if z is even, and `rem 1` otherwise.

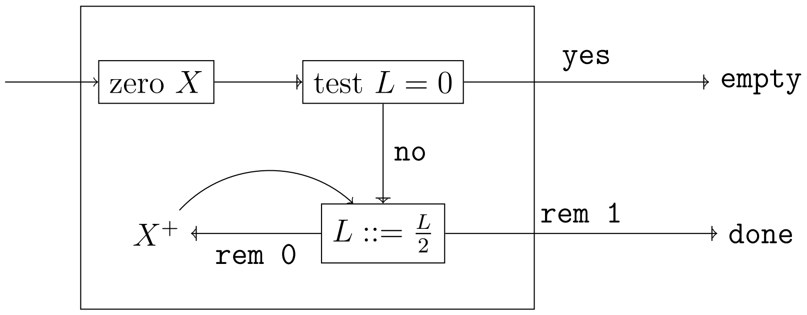


(d) Define a gadget which does the same as the above, but computes it into itself (using a scratch register, which is reset).



(e) Define a gadget that does the following;

- if $X = x$, and $L = 0$, it takes the `empty` exit with $X = L = 0$
- if $X = x$, and $L = \langle\langle y, z \rangle\rangle = 2^y(2z + 1)$, it takes the `done` exit with $X = y$ and $L = z$



zero X
test $L = 0$
yes
empty
no
X^+
rem 0
$L ::= \frac{L}{2}$
rem 1
done