Starter Questions

Feel free to discuss these with your neighbour:

- Do these programs do the same thing?
  
  ```java
  if x<0 then ( x := −x ; z := y−x ; )
  ( if x<0 then x := −x ) ; z := y−x ;
  ```

- What do these programs do?
  
  ```java
  z := 0 ; while y>0 do ( z := z+x ; y := y−1 )
  z := 0 ; while y<0 do ( z := z−x ; y := y+1 )
  ```

- For each pair of sets which set is bigger?
  
  - \( \mathbb{N} \) or \( \mathbb{Z} \)
  - \( \mathbb{N} \) or \( \mathbb{N} \times \mathbb{N} \)
  - \( \mathbb{N} \) or \( \mathbb{R} \)

I will be asking you to do small exercises during the lecture. If you don’t have a pen and paper try and borrow them now.
Lecture 2

The while Language Examples and Coding

COMP11212

Giles Reger

March 2017
Clarifications

Dealing with ambiguity in while programs

- We can use brackets ( and ) to resolve ambiguities
- For example, (if \( x < 0 \) then \( x := -x \)); \( z := y - x \)

What is a state?

- For the purposes of applying the semantics we can just think of a state as a lookup table
- The notation \( s[x \mapsto n] \) copies \( s \) with the value of \( x \) updated to \( n \)
- We will see some examples today

But how is this idea related to state machines?

- It’s the same idea
- In both cases a state represent part of a computation
- In \( < S, s > \) we make the ’rest of the computation’ explicit
Transition System

[ass] \[ \langle x := a, s \rangle \Rightarrow s[x \mapsto A[a]] s \]

[skip] \[ \langle \text{skip}, s \rangle \Rightarrow s \]

[comp\textsuperscript{1}] \[ \frac{\langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle}{\langle S_1; S_2, s \rangle \Rightarrow \langle S'_1; S_2, s' \rangle} \]

[comp\textsuperscript{2}] \[ \frac{\langle S_1, s \rangle \Rightarrow s'}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle} \]

[if\textsuperscript{tt}] \[ \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_1, s \rangle \text{ if } B[b] s = \texttt{tt} \]

[if\textsuperscript{ff}] \[ \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_2, s \rangle \text{ if } B[b] s = \texttt{ff} \]

[while] \[ \langle \text{while } b \text{ do } S, s \rangle \Rightarrow \langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else } \text{skip}, s \rangle \]
Transition System

[ass]  \( <x := a, \; s> \Rightarrow s[x \mapsto A[a]] \; s \)

[skip]  \( <\text{skip}, \; s> \Rightarrow s \)

[comp\textsuperscript{1}]  \( <S_1, \; s> \Rightarrow <S'_1, \; s'> \)

\( <S_1; S_2, \; s> \Rightarrow <S'_1; S_2, \; s'> \)

[comp\textsuperscript{2}]  \( <S_1, \; s> \Rightarrow s' \)

\( <S_1; S_2, \; s> \Rightarrow <S_2, \; s'> \)

[if\textsuperscript{tt}]  \( <\text{if } b \text{ then } S_1 \text{ else } S_2, \; s> \Rightarrow <S_1, \; s> \text{ if } B[b] \; s = \texttt{tt} \)

[if\textsuperscript{ff}]  \( <\text{if } b \text{ then } S_1 \text{ else } S_2, \; s> \Rightarrow <S_2, \; s> \text{ if } B[b] \; s = \texttt{ff} \)

[while]  \( <\text{while } b \text{ do } S, \; s> \Rightarrow <\text{if } b \text{ then } (S; \text{ while } b \text{ do } S) \text{ else } \text{skip}, \; s> \)
Transition System

[ass] \( < x := a, s > \Rightarrow s[x \mapsto A[a]] s \)

[skip] \( < \text{skip}, s > \Rightarrow s \)

[comp\(^1\)] \[ < S_1, s > \Rightarrow < S'_1, s' > \]
\[ < S_1; S_2, s > \Rightarrow < S'_1; S_2, s' > \]

[comp\(^2\)] \[ < S_1, s > \Rightarrow s' \]
\[ < S_1; S_2, s > \Rightarrow < S_2, s' > \]

[if\(^{tt}\)] \[ < \text{if } b \text{ then } S_1 \text{ else } S_2, s > \Rightarrow < S_1, s > \quad \text{if } B[b] s = \text{tt} \]

[if\(^{ff}\)] \[ < \text{if } b \text{ then } S_1 \text{ else } S_2, s > \Rightarrow < S_2, s > \quad \text{if } B[b] s = \text{ff} \]

[while] \[ < \text{while } b \text{ do } S, s > \Rightarrow \]
\[ < \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip, } s > \]
Transition System

[\text{ass}] \quad < x := a, \ s > \ \Rightarrow \ s[x \mapsto A[a]] s

[\text{skip}] \quad < \text{skip}, \ s > \ \Rightarrow \ s

[\text{comp}^1] \quad \frac{< S_1, \ s > \ \Rightarrow \ < S'_1, \ s' >}{< S_1 ; S_2, \ s > \ \Rightarrow \ < S'_1 ; S_2, \ s' >}

[\text{comp}^2] \quad \frac{< S_1, \ s > \ \Rightarrow \ s'}{< S_1 ; S_2, \ s > \ \Rightarrow \ < S_2, \ s' >}

[\text{if}^\text{tt}] \quad < \text{if } b \text{ then } S_1 \text{ else } S_2, \ s > \ \Rightarrow \ < S_1, \ s > \quad \text{if } B[b] s = \text{tt}

[\text{if}^\text{ff}] \quad < \text{if } b \text{ then } S_1 \text{ else } S_2, \ s > \ \Rightarrow \ < S_2, \ s > \quad \text{if } B[b] s = \text{ff}

[\text{while}] \quad < \text{while } b \text{ do } S, \ s > \ \Rightarrow \\
\quad \quad \quad < \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, \ s >
Transition System

[ass]  \( < x := a, s > \Rightarrow s[x \mapsto A[a]] s \)

[skip]  \( < \text{skip}, s > \Rightarrow s \)

[comp\(^1\)]  
\[
\frac{< S_1, s > \Rightarrow < S'_1, s' >}{< S_1; S_2, s > \Rightarrow < S'_1; S_2, s' >}
\]

[comp\(^2\)]  
\[
\frac{< S_1, s > \Rightarrow s'}{< S_1; S_2, s > \Rightarrow < S_2, s' >}
\]

[if\(^{\text{tt}}\)]  
\( < \text{if } b \text{ then } S_1 \text{ else } S_2, s > \Rightarrow < S_1, s > \text{ if } B[b] s = \text{tt} \)

[if\(^{\text{ff}}\)]  
\( < \text{if } b \text{ then } S_1 \text{ else } S_2, s > \Rightarrow < S_2, s > \text{ if } B[b] s = \text{ff} \)

[while]  
\( < \text{while } b \text{ do } S, s > \Rightarrow < \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else } \text{skip}, s > \)
What do while programs compute?

Answer: Functions in $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$

Because they take $n$ variables with given values and set $m$ variables to some (new) values.

But as we see later

- Not all such functions (we see this in a few weeks)
- And we could equally say they compute functions in $\mathbb{N} \rightarrow \mathbb{N}$

We now want to convince ourselves that this is *good enough* i.e. that we do not need anything else
The idea is to show that all functions $A \to B$ can be mapped to some function in $\mathbb{N} \to \mathbb{N}$ for any domains $A$ and $B$.

For example

- if $A$ were the set of all integer arrays and $B$ was integers then we could capture the max function on arrays
- if $A$ and $B$ were both arrays then we could capture sorting
- if $A$ were the set of graphs and $B$ the set of node lists then we could capture Dijkstra’s algorithm

To do this we will show how other domains can be coded into $\mathbb{N}$
Mapping $\mathbb{Z}$ to $\mathbb{N}$

I missed this bit out of the notes because I missed the implicit assumption!

**Lemma**

There is a bijection between $\mathbb{N}$ and $\mathbb{Z}$

Reminder:

- $f$ is injective if $\forall x, y : f(x) = f(y) \rightarrow x = y$
  
  Every input has a unique output

- $f$ is surjective if $\forall x, \exists y : f(y) = x$
  
  Every possible output has an input that generates it

- $f$ is a bijection if it is injective and surjective
  
  Which means there must be an inverse
Let us define $\beta(x) : \mathbb{Z} \to \mathbb{N}$ as

$$\beta(x) = \begin{cases} 
2x & \text{if } x \geq 0 \\
-2x - 1 & \text{otherwise}
\end{cases}$$

For example,

$$\beta(0) = 0 \quad \beta(1) = 2 \quad \beta(-1) = 1 \quad \beta(5) = 10 \quad \beta(-3) = 5$$

i.e. it maps positive numbers to even numbers and negative to odd

This is a bijection
Mapping $\mathbb{Z}$ to $\mathbb{N}$

We can compute $\beta$

$$\text{if } x \geq 0 \text{ then } z := 2 \times x \text{ else } z := (-2 \times x) - 1$$

We can compute its inverse $\beta^{-1}$ (on inputs in $\mathbb{N}$)

$$r := x; \quad z := 0;$$
$$\text{while } (2 \leq r) \text{ do } \begin{cases} \quad z := z + 1; \quad r = r - 2; \\ \text{if } r=1 \text{ then } z := -z - 1 \end{cases}$$

We can transform any program using $\mathbb{Z}$ into one using $\mathbb{N}$ only
Dealing with Pairs

Next we show that we can code arbitrary pairs of numbers as single numbers (in $\mathbb{N}$ due to the previous argument)

The following function $\phi$ maps pair $(n, m)$ to a unique number in $\mathbb{N}$

$$\phi(n, m) = 2^n(2m + 1) - 1$$

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 . . .</td>
</tr>
<tr>
<td>0</td>
<td>0 2 4 . . .</td>
</tr>
<tr>
<td>m</td>
<td>1 5 9 . . .</td>
</tr>
<tr>
<td>2</td>
<td>3 11 19 . . .</td>
</tr>
<tr>
<td>...</td>
<td>... ... ... ...</td>
</tr>
</tbody>
</table>

$\phi(2, 3) = 27 = 11011_2$

$\phi(3, 4) = 71 = 1001111_2$

The function $\phi$ is a bijection
Lists (of integers) can be recursively defined as

\[ \text{list} = [] \mid n :: \text{list} \]

We can code lists using a recursively defined function \( \varphi \)

\[
\begin{align*}
\varphi([]) &= 0 \\
\varphi(n :: l) &= 2^n(2\varphi(l) + 1) = \phi(n, \varphi(l)) + 1
\end{align*}
\]

The +1 at the end is to give 'space' for the empty list.

This trick of using \( \phi \) to define \( \varphi \) means we just need a good \( \phi \).
What is missing in the above is the point that operations on coded data structures become arithmetic.

We can write a while program to:

- construct a pair
- extract one half of a pair
- extract the head or tail of a list
- append two lists
- reverse a list

etc

All using the arithmetic operations available in while.
Everything can be written down as a list or a pair e.g.

- An object in Java is a list of the fields that it contains
- Characters can be mapped directly to numbers
- Strings are just lists of characters
- A graph is a list of pairs

So we can just deal with programs that compute functions in \( \mathbb{N} \rightarrow \mathbb{N} \) and argue that we can also compute functions on any other data structures.
Reflective Questions

- What does this program do?
  
  ```
  while true do x := x+1
  ```

- Can we encode \( \mathbb{R} \) in \( \mathbb{N} \)? The real numbers between 0 and 1? The real numbers between 0 and 0.5? . . .

- How do our coding functions relate to hash functions from cryptography? What is the pigeonhole principle?

- What if we wanted \((n, m)\) and \((m, n)\) to have the same coding? How could we extend such a coding function to code sets so that the order did not effect the coding?

- How can we use the idea of prime factorisation in coding?