Do you remember?

- To **compute** a **function** we run a **program**
- A programming language needs a **syntax** and **semantics**
- We use brackets to deal with ambiguity in **while**
- We can **extend the syntax** of **while** via definitions

Something to think about:

Imagine we had a language which consisted of three kinds of symbols

1. The numbers 0 to 27 e.g. ‘4’ or ‘14’
2. The expressions $a + b$ for $a, b$ in $\{0, \ldots, 18\}$ e.g. ‘4+9’ or ‘13+1’
3. The expressions $(a + b) + c$ for $a, b, c$ in $\{0, \ldots, 9\}$ e.g. ‘(4+9)+1’

Can we write a DFA to capture **evaluation**? e.g. the strings

- ‘(4+9)+1’.’11+1’.’14’ , ‘(1+1)+1’).’2+1’.’3’ and ‘0+0’.’0’
- should be accepted but the strings

- ‘(8+2’)’11’ , ‘0+0’.’1’ and ‘1+1’.’1+1’.‘2’

should not.

**All exercises for Examples Classes will appear on the course website**
Admin Things

The structure for coursework is the same as in Part I

The exercises for Examples Classes will appear on the course website (they are not at the back of notes)

Solutions will be posted in the same way as for Part I

Lecture slides will be posted on the website after the lecture

The website also contains some tools you can use to explore the material
Every language needs:

- a well-defined **syntax** i.e. the form that programs are allowed to take
- a well-defined **semantics** i.e. what a program means

Yesterday we saw the **syntax** of `while` e.g. the grammar

\[
S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S
\]

\[
b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2
\]

\[
a ::= x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2
\]

Today we will look at its **operational semantics**.
Operational Semantics

To define the semantics of the \texttt{while} language, we need ways to

1. Represent the current values of variables i.e. the \texttt{state}. For this we just need to be able to \texttt{lookup} and \texttt{update} the value for a variable.

2. Evaluate the arithmetic and logical expressions (here we define what they \texttt{denote} and then apply standard arithmetic and logic)

3. Describe how program statements \texttt{transform state}. To do this will we introduce a relation $\Rightarrow$ on programs and states such that

$$\langle S\sigma \rangle \Rightarrow \langle S', \sigma' \rangle$$

if we execute program $S$ in state $\sigma$ to give a new program $S'$ and new state $\sigma'$.

So far I have not said anything explicit about \texttt{while}. 

Giles Reger

Lecture 0

March 2018
To define the semantics of the while language, we need ways to

1. Represent the current values of variables i.e. the state. For this we just need to be able to lookup and update the value for a variable.

2. Evaluate the arithmetic and logical expressions (here we define what they denote and then apply standard arithmetic and logic)

3. Describe how program statements transform state. To do this will we introduce a relation $\Rightarrow$ on programs and states such that

$$\langle S\sigma \rangle \Rightarrow \langle S', \sigma' \rangle \Rightarrow \ldots \Rightarrow \sigma''$$

if we execute program $S$ in state $\sigma$ to give a new program $S'$ and new state $\sigma'$.

So far I have not said anything explicit about while.
Pause for Thought

Before we look at while, are there other languages that we could consider the operational semantics of?

**Arithmetic**: We saw a bit of this before. But we can do this more generally for formulae containing variables e.g.

\[
\langle x + 0, \sigma \rangle \Rightarrow \langle x, \sigma \rangle \quad \langle x + y, \sigma \rangle \Rightarrow n \text{ if } n = \sigma(x) + \sigma(y)
\]

**Regular Expressions**: Use the idea of derivatives to rewrite a pair consisting of an expression and a word e.g.

\[
\langle ar, x\tau \rangle \Rightarrow \langle r, \tau \rangle \text{ if } a = x \quad \begin{array}{c}
\langle s, a \rangle \Rightarrow \langle s', \epsilon \rangle \quad \langle t, a \rangle \Rightarrow \langle t', \epsilon \rangle \\
\langle s \mid r, a\tau \rangle \Rightarrow \langle s' \mid t', \tau \rangle
\end{array}
\]
Representing State

We abstract state as a mapping between variables and integers

We define the possible states as the following space of total functions

\[ \text{State} = \text{Var} \to \mathbb{Z} \]

Consider a state as a bit of indexible infinite memory (a lookup table).

We write these as, e.g. \([x \mapsto 1, y \mapsto 2] \). We write \([\] \) for the empty state.

We will write \( \sigma[x \mapsto n] \) for the state \( \sigma \) with the value for \( x \) updated to \( n \)

This is a reasonable abstraction but what issues does this present for the argument that while is a good model of computation?
Evaluating Expressions

We define a function that takes any arithmetic expression and a state and evaluates the expression. This defines what such expressions denote.

\[ A : \text{AExp} \to \text{State} \to \mathbb{Z} \]

\[ A[n] \ s = N[n] \]

\[ A[x] \ s = s(x) \]

\[ A[a_1 + a_2] \ s = A[a_1] \ s + A[a_2] \ s \]

\[ A[a_1 \times a_2] \ s = A[a_1] \ s \times A[a_2] \ s \]

\[ A[a_1 - a_2] \ s = A[a_1] \ s - A[a_2] \ s \]

What assumptions does this make about the state?
Evaluating Expressions

We define a function that takes any arithmetic expression and a state and evaluates the expression. This defines what such expressions denote.

\[ A : \text{AExp} \to \text{State} \to \mathbb{Z} \]

\[ A[n] \ s \ = \ \mathcal{N}[n] \]
\[ A[x] \ s \ = \ s(x) \]
\[ A[a_1 + a_2] \ s \ = \ A[a_1] \ s + A[a_2] \ s \]
\[ A[a_1 \times a_2] \ s \ = \ A[a_1] \ s \times A[a_2] \ s \]
\[ A[a_1 - a_2] \ s \ = \ A[a_1] \ s - A[a_2] \ s \]

What assumptions does this make about the state? We assume that all variables have an initial value of 0.
Evaluating Expressions

We do the same for logical expressions.

\[ B : \text{BExp} \rightarrow \text{State} \rightarrow \text{B} \]

\[ B[\text{true}] s = \text{tt} \]
\[ B[\text{false}] s = \text{ff} \]

\[ B[a_1 = a_2] s = \begin{cases} \text{tt} & \text{if } A[a_1] s = A[a_2] s \\ \text{ff} & \text{if } A[a_1] s \neq A[a_2] s \end{cases} \]

\[ B[a_1 \leq a_2] s = \begin{cases} \text{tt} & \text{if } A[a_1] s \leq A[a_2] s \\ \text{ff} & \text{if } A[a_1] s > A[a_2] s \end{cases} \]

\[ B[\neg b] s = \begin{cases} \text{tt} & \text{if } B[b] s = \text{ff} \\ \text{ff} & \text{if } B[b] s = \text{tt} \end{cases} \]

\[ B[b_1 \land b_2] s = \begin{cases} \text{tt} & \text{if } B[b_1] s \text{ and } B[b_2] s \\ \text{ff} & \text{if } \text{not } (B[b_1] s \text{ and } B[b_2] s) \end{cases} \]
The next slide introduces a transition system for transforming configurations of the form \( \langle S, \sigma \rangle \)

into either another configuration or a final state, where \( S \) is a program and \( \sigma \) is a state

We can think of each possible configuration as a state in an infinite state machine. The following system describes an infinite set of transitions. This big (deterministic) state machine represents all possible computations.

It is called structural because the rules follow the structure of the program

It is called small-step as it defines each step
Transition System

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

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

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

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

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

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

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

[ass] \[ < x := a, \sigma > \Rightarrow \sigma[x \mapsto A[a] \sigma] \]

[skip] \[ < \text{skip}, \sigma > \Rightarrow \sigma \]

[comp\,]^1 \[ \begin{array}{c} < S_1, \sigma > \Rightarrow < S'_1, \sigma' > \\ < S_1; S_2, \sigma > \Rightarrow < S'_1; S_2, \sigma' > \end{array} \]

[comp\,]^2 \[ \begin{array}{c} < S_1, \sigma > \Rightarrow \sigma' \\ < S_1; S_2, \sigma > \Rightarrow < S_2, \sigma' > \end{array} \]

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

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

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

[ass] \[< x := a, \sigma > \Rightarrow \sigma[x \mapsto A[a]] \sigma]\]

[skip] \[< \text{skip}, \sigma > \Rightarrow \sigma]\]

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

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

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

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

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

[ass] \[< x := a, \sigma > \Rightarrow \sigma[x \mapsto A[a]] \sigma]\]

[skip] \[< \text{skip}, \sigma > \Rightarrow \sigma]\]

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

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

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

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

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

**[ass]**
\[< x := a, \sigma > \Rightarrow \sigma[x \mapsto A[a] \sigma]\]

**[skip]**
\[< \text{skip}, \sigma > \Rightarrow \sigma\]

**[comp\textsuperscript{1}]**
\[
\begin{align*}
\frac{< S_1, \sigma > \Rightarrow < S_1', \sigma' >}{< S_1; S_2, \sigma > \Rightarrow < S_1'; S_2, \sigma' >}
\end{align*}
\]

**[comp\textsuperscript{2}]**
\[
\begin{align*}
\frac{< S_1, \sigma > \Rightarrow \sigma'}{< S_1; S_2, \sigma > \Rightarrow < S_2, \sigma' >}
\end{align*}
\]

**[if\textsuperscript{tt}]**
\[< \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma > \Rightarrow < S_1, \sigma > \text{ if } B[b] \sigma = \text{tt}\]

**[if\textsuperscript{ff}]**
\[< \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma > \Rightarrow < S_2, \sigma > \text{ if } B[b] \sigma = \text{ff}\]

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

[ass] \[< x := a, \sigma > \Rightarrow \sigma[x \mapsto A[a] \sigma]\]

[skip] \[< \text{skip}, \sigma > \Rightarrow \sigma\]

[comp\textsuperscript{1}] \[
\frac{< S_1, \sigma > \Rightarrow < S_1', \sigma' >}{< S_1; S_2, \sigma > \Rightarrow < S_1'; S_2, \sigma' >}
\]

[comp\textsuperscript{2}] \[
\frac{< S_1, \sigma > \Rightarrow \sigma'}{< S_1; S_2, \sigma > \Rightarrow < S_2, \sigma' >}
\]

[if\textsuperscript{tt}] \[< \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma > \Rightarrow < S_1, \sigma > \text{ if } B[b] \sigma = \text{tt}\]

[if\textsuperscript{ff}] \[< \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma > \Rightarrow < S_2, \sigma > \text{ if } B[b] \sigma = \text{ff}\]

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

\[
< z := 1, \emptyset > \Rightarrow [z \mapsto 1] \quad \text{Rule [ass]}
\]

\[
< x := x + 1, [x \mapsto 2] > \Rightarrow [x \mapsto 3] \quad \text{Rule [ass]}
\]

\[
< x := x + y, [x \mapsto 2] > \Rightarrow [x \mapsto 2] \quad \text{Rule [ass]}
\]

\[
\text{< if } (0 \leq x) \text{ then skip else } x := 0 - x, [x \mapsto -17] > \\
\Rightarrow \quad < x := 0 - x, [x \mapsto -17] > \quad \text{Rule [iff]}
\Rightarrow \quad [x \mapsto 17] \quad \text{Rule [ass]}
\]
Another Example

\[
\begin{align*}
< x := 0; \text{while } (x > 0) (x := x - 1), [] > \\
\Rightarrow < \text{while } (x > 0) (x := x - 1), [x \mapsto 0] > & \quad \text{Rule [ass]} \\
\Rightarrow < \text{if } (x > 0) \text{ then } (x := x - 1; \\
& \quad \text{while } (x > 0) (x := x - 1)) \text{ else } \text{skip}, [x \mapsto 0] > & \quad \text{Rule [while]} \\
\Rightarrow < \text{skip}, [x \mapsto 0] > & \quad \text{Rule [if]} \\
\Rightarrow [x \mapsto 0] & \quad \text{Rule [skip]}
\end{align*}
\]
Another Another Example

\[< x := 1; \text{while} (x > 0) (x := x - 1), [] > \]
\[\Rightarrow < \text{while} (x > 0) (x := x - 1), [x \mapsto 1] > \text{ Rule [ass]} \]
\[\Rightarrow < \text{if} (x > 0) \text{ then } (x := x - 1; \text{while} (x > 0) (x := x - 1)), [x \mapsto 1] > \text{ Rule [while]} \]
\[\Rightarrow < x := x - 1; \text{while} (x > 0) (x := x - 1), [x \mapsto 1] > \text{ Rule [if]} \]
\[\Rightarrow < \text{while} (x > 0) (x := x - 1), [x \mapsto 1] > \text{ Rule [ass]} \]
\[\Rightarrow \ldots \]
\[\Rightarrow [x \mapsto 0] \]
The Derived Composition Rules

I never explicitly write out the composition rules because it means the derivations become non-linear, which is a pain.

Instead I use implicit derived rules that combine a composition rule with another rule. For example, the following rule for assignment:

\[
[\text{ass-comp}] < x := a; S, \sigma > \Rightarrow < S, s[x \mapsto A[a] \sigma] >
\]

It should be easy to see that this rule can be proved sound with respect to the original rules.

For example, we can also write

\[
\langle (\text{if } b \text{ then } S_1 \text{ else } S_2); S_3, \sigma \rangle \Rightarrow \langle S_1; S_3, \sigma \rangle \text{ if } B[b] \sigma = \text{tt}
\]
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 extending the language with more kinds of structures does not increase expressiveness.
From $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$ to $\mathbb{N} \rightarrow \mathbb{N}$ (and back again)

Today and next week we will show that

1. We can directly relate functions in $\mathbb{Z} \rightarrow \mathbb{Z}$ and functions in $\mathbb{N} \rightarrow \mathbb{N}$
2. We can lift this to arbitrary sequences

and thus, any function in $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$ can be represented by one in $\mathbb{N} \rightarrow \mathbb{N}$.

I use the term relate as we will introduce bijections meaning that we can go in either direction.
Mapping $\mathbb{Z}$ to $\mathbb{N}$

Lemma

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

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 numbers

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

We can compute $\beta$

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

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

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

We can transform any program using $\mathbb{Z}$ into one using $\mathbb{N}$ only
Take Home Points

To define operational semantics we need to define the states and transitions between them

while programs do not necessarily terminate

while programs compute functions in $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$

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