Starter Questions

Feel free to discuss these with your neighbour:

- Why do we need semicolons in Java?
- What does this print
  `System.out.println(new Integer(100) == new Integer(100));`
- How many symbols do you need in propositional logic?
- Can you implement multiplication using addition, conditions and loops?
- What is the smallest set of things a programming language needs for it to be able to do anything?!
- Did you pick up the Handout from SSO?
Lecture 1

The while Language

COMP11212

Giles Reger

March 2017
A program just describes some things we need to do:

- Here’s a nail and a hammer; when I nod my head, you hit it
- Go to the shops and get 2 pints of milk. If they have eggs get 6
- Put the value of $x$ in $y$ and the value of $y$ in $x$

English (and natural language in general) is highly ambiguous. We need a precise language for giving instructions.

But how precise is good enough?
How to Define a Programming Language

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

All languages have a well-defined syntax.

Not all languages have well-defined **semantics**. Very few have a formal **semantics**, which is what we see in this course.

Instead, they rely on informal descriptions or reference implementations. Let’s take a quick look at some examples.
Meaning of a program $\equiv$ what function it computes

There are three main approaches:

1. **Operational**: define what it means to interpret/execute programs
2. **Axiomatic**: define a program by what it provably does
3. **Denotational**: transform programs into some other metalanguage

We will see a bit of all three in this course

\textit{ALGOL 68 was the first (and possibly one of the last) major language for which a full formal definition was made before it was implemented.}

\textit{C.H.A. Koster}
The while language

Is **small** enough to understand and be useful

Is **powerful** enough to capture any other language we might want

We will present its **syntax** and **operational semantics**
The syntax can be given as a grammar (this is the usual way)

\[
S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \\
b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \\
a ::= x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2
\]

Why is this ambiguous?

In the above \( n \) stands for numbers (e.g. integers \( \mathbb{Z} \)). Later we see that this is sufficient (what does that mean?).
Example

What does this program do?

\[ x := 1; \quad y := 5; \]
\[ \textbf{while} \; (y > 0) \; \{ \; x := x + x; \quad y := y - 1; \; \} \]

We assume variables can have some initial values: this is \textit{input}.
We assume we can read the variables at the end: this is \textit{output}. 
What does this program do?

\[ x := 1; \]
\[ \textbf{while } (y > 0) (x := x + x; \ y := y - 1); \]

We assume variables can have some initial values: this is input
We assume we can read the variables at the end: this is output
What if we want to write

if \( b \) then \( S \)

when we only have

if \( b \) then \( S_1 \) else \( S_2 \)
We can define syntactic extensions in terms of existing syntax

\[ \text{[if } b \text{ then } S] \equiv \text{[if } b \text{ then } S \text{ else skip]} \]

This idea should be familiar.

In propositional logic we only need \( \land \) and \( \neg \), i.e.

\[ (A \lor B) \equiv \neg(\neg A \land \neg B) \]

In fact, we only need \textit{Nand} to be functionally complete. There is a related notion of \textit{Turing complete} here, which we will meet later.
Another Program

We will use this program in examples a lot, so you will be familiar with it by the end of the course.

\[
\begin{align*}
    r & := x; \\
    d & := 0; \\
    \textbf{while } y \leq r \textbf{ do } (d := d+1; r := r-y)
\end{align*}
\]

This program computes the function

\[f(x, y) = (d, r) = (x \div y, x \mod y)\]

whenever \(y > 0 \land x \geq 0\)
Semantics

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

- Represent the current values of variables i.e. the state
- Evaluate the arithmetic and logical expressions
- Describe how program statements transform state
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.

We will write these as, for example, \([x \mapsto 1, y \mapsto 2]\)

We will write \(s[x \mapsto n]\) for the state \(s\) 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?
We define a function that takes any arithmetic expression and a state and evaluates the expression. This defines what such expressions denote.

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

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

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

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

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

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

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.

\[ A : \text{AExp} \rightarrow \text{State} \rightarrow \mathbb{Z} \]
\[ 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 \[ a_1 \ast a_2 \] \ s = A \[ a_1 \] \ s \ast 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.

\[ B : \text{BExp} \rightarrow \text{State} \rightarrow \{\text{tt, ff}\} \]

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

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

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

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

\[ B[b_1 \land b_2] \ s = \begin{cases} \text{tt} & \text{if } B[b_1] \ s \ \text{and} \ B[b_2] \ s \\ \text{ff} & \text{if } \neg (B[b_1] \ s \ \text{and} \ B[b_2] \ s) \end{cases} \]
The next slide introduces a transition system for transforming configurations of the form

\[ \langle S, s \rangle \]

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

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

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

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

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

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

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

\[\text{[while]} \quad < \text{while } b \text{ do } S, \ s > \ \Rightarrow \]
\[\quad < \text{if } b \text{ then } (S; \text{ while } b \text{ do } S) \text{ else } \text{skip}, \ s >\]
< x := 2; x := 1, [x ↦ 0] >

< if (0 ⩽ x) then skip else x := 0 − x, [x ↦ −17] >

< while (x > 0) (x := x − 1), [] >

< while (x > 0) (x := x − 1), [x ↦ 1] >
Reflective Questions

- How many while programs are there that result in a state \( x \mapsto 1 \)? (Hint, it’s more than one)
- How many different while programs are there in general?
- What would our idea of state look like for a language like Java? What else do we need to keep track of?
- What would we have to deal with if we used a more realistic set of numbers, for example 32-bit integers?