Syntax and semantics

Why do we need these? Don’t we already know a lot about programming?

We are NOT going to define or redefine programming languages. We will introduce notations to represent abstract algorithms, not executable codes.

Your programs need to capture the notions of atomicity, non-determinism, fairness etc. JAVA, C etc don’t have it! They need to be implemented!

Guarded actions (Dijkstra)

Sequential actions       \( S0; S1; S2; \ldots; Sn \)
Alternative constructs  \( \text{if } \ldots \ldots \ldots \ldots \text{fi} \)
Repetitive constructs   \( \text{do } \ldots \ldots \ldots \text{od} \)

What is a guard?
**Alternative construct**

\[
\text{if} \quad G_0 \sqsupseteq S_0 \\
G_1 \sqsupseteq S_1 \\
G_2 \sqsupseteq S_2 \\
\ldots \\
G_n \sqsupseteq S_n \\
\text{fi}
\]

When no guard is true, **skip** (do nothing). When multiple guards are true, the choice of the action to be executed is completely arbitrary.

**Repetitive construct**

\[
\text{do} \quad G_0 \sqsupseteq S_0 \\
G_1 \sqsupseteq S_1 \\
G_2 \sqsupseteq S_2 \\
\ldots \\
G_n \sqsupseteq S_n \\
\text{od}
\]

Keep executing the actions until all guards are false. When multiple guards are true, the choice of the action is arbitrary.
Example 1

program uncertain;
define x : integer;
initially x = 0
do x < 4 ▷ x := x + 1
   x = 3 ▷ x := 0
od

Will the program terminate?
Determinism vs. Non-determinism

Example 2
Determinism is too rigid in distributed systems. Consider a process with k channels $c_0 \ldots c_{k-1}$.

\[
\text{define } x : \text{array} [0..k-1] \text{ of boolean}
\]
\[
\text{initially all channels are empty}
\]
\[
\text{do } \neg \text{empty (}c_0\text{)} \quad \text{send ack along } c_0
\]
\[
\neg \text{empty (}c_1\text{)} \quad \text{send ack along } c_1
\]
\[
\quad \cdot
\]
\[
\neg \text{empty (}c_{k-1}\text{)} \quad \text{send ack along } c_{k-1}
\]
\[
\text{od}
\]

How can determinism (i.e. a fixed polling order) affect the outcome in the above program?
**Atomicity (also called granularity)**

Atomic actions are indivisible. *All or nothing* property

```
    do  red  x:=0
        blue  x:=7
    od
```

If messages continuously keep coming along the two channels, then what can be possible values of x?

- 0 or 7? Hard to say!
- Any value between 0 and 7! It is possible!
- Depends on the atomicity of the assignment operation!
More examples of atomic actions

\[
\begin{align*}
\text{define} & \quad b : \text{boolean} \\
\text{initially} & \quad b = \text{true} \\
\text{do} & \quad b \quad \quad \quad \quad \text{send message m to Q} \\
& \quad \neg \text{empty} (R, P) \quad \quad \text{receive message;} \\
& \quad \quad \quad \quad \quad b := \text{false} \\
\text{od}
\end{align*}
\]

Suppose it takes 15 seconds to send the message. After 5 seconds, P receives a message from R. Will it stop sending the remainder of the message? \textbf{NO.}

Atomic actions are indivisible. All statements $G_i \dashv S_i$ are atomic unless specified otherwise.
Coarse-grain vs. fine-grain atomicity

Every process executes $G \in S$.

Let $G = f$ (states of all neighbors).
How will the actions of the processes be interleaved?
There are many choices, and the outcome may be different for each choice.

Understand the differences between coarse-grain (stronger model) and fine-grain atomicity (weaker model).

How to implement the various grains of atomicity?
Scheduling and fairness

Whenever multiple guards are true, there is more than one action to choose from. The choice of these possible alternatives is determined by **fairness**. Fairness is a property of the **scheduler**, and it has important effects on the behavior of a program.

**Broad classifications**

0. Unfair scheduler
1. Fair scheduler
   a. Unconditional fairness
   b. Weak fairness
   c. Strong fairness

What are the differences???
Scheduling and fairness (continued)

Consider the following program:

```plaintext
program  test
define   x  : integer {initial value unknown}

  do  true  ‡  x : = 0
      x=0  ‡  x : = 1
      x=1  ‡  x : = 2
  od
```

An unfair scheduler *may never* schedule the second (or the third actions). So, \( x \) may always be equal to zero.

An unconditionally fair scheduler will *eventually* give every statement a chance to be executed without checking their eligibility. (Example: process scheduler in a multiprogrammed OS.)
**Weakly fair scheduler**

A scheduler is weakly fair, when it eventually executes every guarded action whose guard becomes true, and remains true thereafter.

```
program test
define x : integer {initial value unknown}

do true ⟷ x := 0
   x=0 ⟷ x := 1
   x=1 ⟷ x := 2
od
```

So, a weakly fair scheduler will eventually execute the second action, but may never execute the third action. Why?

It will take a strongly fair scheduler to guarantee the execution of the third action.
**Strongly fair scheduler**

A scheduler is strongly fair, if it executes every guarded statement whose guard is true "infinitely often".