Introduction to semantics

- Let's introduce our semantics by means of an example

- First, let's decide what the semantics will be used for in our example:
  - To ensure that the program is correct (this is called *verification*)
  - To make sure the program is well-designed
  - To explain the program to others
  - To calculate time and memory utilisation
  - To understand how the program manages memory (in particular, how it does *garbage collection*)

- Let's choose the first goal, namely correctness

When is a program correct?

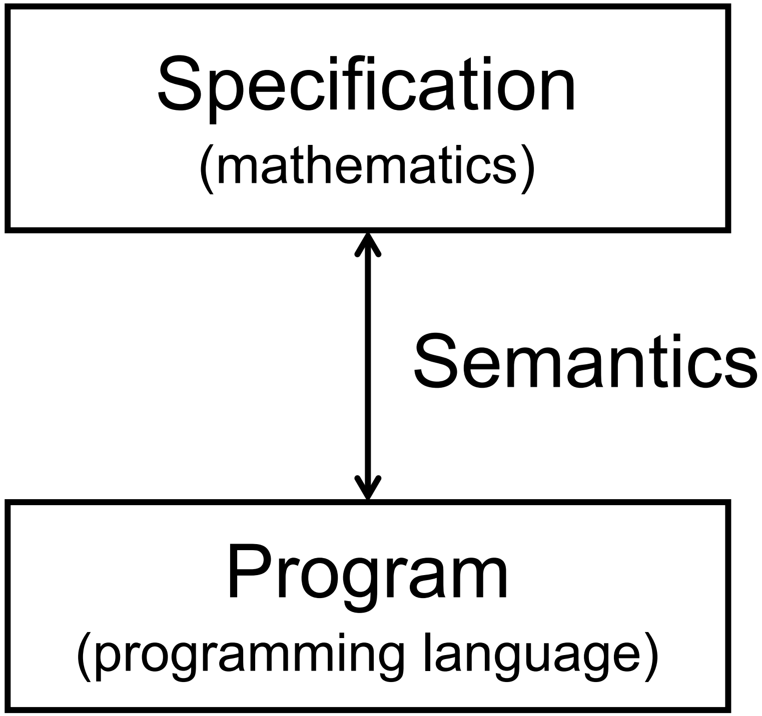
- “A program is correct when it does what we want it to”
- How can we be sure?

- There are two starting points:
  - **The program’s specification**: a mathematical definition of the result of the program as a function of the input
  - **The language semantics**: a precise mathematical model of how a program executes

- We need to prove that the program satisfies the specification, when it executes according to the semantics

The three pillars

- The specification:
  what we want
- The program:
  what we have
- The semantics connects these two: proving that what we have executes according to what we want



Example: correctness of factorial

- The specification of {Fact N} (mathematics)

  $$0! = 1$$
  $$n! = n \times ((n\text{-}1)!) \quad \text{when} \quad n>0$$

- The program (programming language)

```
fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end
```

- The semantics connects the two

Mathematical induction

- To make this proof for a recursive function we need to use mathematical induction
  - A recursive function calculates on a recursive data structure, which has a base case and a general case
  - We first show the correctness for the base case
  - We then show that if the program is correct for a general case, it is correct for the next case
- For integers, the base case is usually 0 or 1, and the general case *n-1* leads to the next case *n*
- For lists, the base case is usually nil or a small list, and the general case T leads to the next case H|T

The inductive proof

- We must show that {Fact N} calculates $n!$ for all $n \geq 0$

- Base case: $n=0$
  - The specification says: $0!=1$
  - The execution of {Fact 0}, *using the semantics*, gives {Fact 0}=1
    - *It's correct!*
- General case: $(n-1) \rightarrow n$
  - The specification says: $n! = n \times (n-1)!$
  - The execution of {Fact N}, *using the semantics*, gives {Fact N} = N*{Fact N-1}
    - We assume that {Fact N-1}=$(n-1)!$
    - We assume that the language correctly implements multiplication
    - Therefore: {Fact N} = N*{Fact N-1} = $n \times (n-1)! = n!$
    - *It's correct!*

- Now we just need to understand the magic words "*using the semantics*"!

How to execute a program *using the semantics*

- We execute the program using the semantics by following two steps

- First, we translate the program into kernel language
  - The kernel language is a simple language that has all essential concepts
  - All programs in the practical language can be translated into kernel language
  - ➞ We translate the definition of Fact into kernel language

- Second, we execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise mathematical definition
  - ➞ We execute the call {Fact 0 R} on the abstract machine

Executing Fact using the semantics

- We need to execute both {Fact 0} and {Fact N} using the semantics
- First we translate the definition of Fact into kernel language:

```
proc {Fact N R}
    local B in
        B=(N==0)
        if B then R=1
        else local N1 R1 in
            N1=N-1
            {Fact N1 R1}
            R=N*R1
          end
        end
    end
end
```

Execution of {Fact 0} (1)

- Let's first look at the function call {Fact 0}
- We execute the procedure call {Fact N R} where N=0
- We need a memory σ and an environment *E*:

  σ = {*fact*=(**proc** {$ N R} … **end**,{Fact→*fact* }), *n*=0, *r*}
  E = {Fact→*fact*, N→*n*, R→*r* }

- Here is what we will execute:

  {Fact N R}, E, σ

Execution of {Fact 0} (2)

- To execute {Fact N R} we replace it by the procedure body

- The instruction:

  {Fact N R}, {Fact→*fact*, N→*n*, R→*r* }, σ

  is replaced by the instruction:

```
local B in
    B=(N==0)
    if B then R=1 else … end
end, {Fact→fact, N→n, R→r }, σ
```

Execution of {Fact 0} (3)

- To execute the **local** instruction:

  local B **in**
  B=(N==0)
  if B **then** R=1 **else** … **end**
  end, {Fact→*fact*, N→*n*, R→*r* }, σ

  we do two operations:
  - We extend the memory with a new variable *b*
  - We extend the environment with {B → *b*}

- We then replace the instruction by its body:

  B=(N==0)
  if B **then** R=1 **else** … **end**,
  {Fact→*fact*, N→*n*, R→*r*, B → *b*}, σ∪{*b*}

Execution of {Fact 0} (4)

- We now do the same for:

  B=(N==0)

  and:

  if B **then** R=1 **else** … **end end**

- This will first bind *b*=**true** and then bind *r=1*
- This completes the execution of {Fact 0}

- We have executed {Fact 0} with the semantics and shown that the result is 1
- To complete the proof, we still have to show that the result of {Fact N} is the same as N*{Fact N-1}

We have proved the correctness of Fact

- Let's recapitulate the approach
- Start with the specification and program of Fact
  - We want to prove that the program satisfies the specification
  - Since the function is recursive, our proof uses mathematical induction

- We need to prove the base case and the general case:
  - Prove that {Fact 0} execution gives 1
  - Prove that {Fact N} execution gives N*{Fact N-1}

- We prove both cases using the semantics and the Fact program
  - To use the semantics, we first translate Fact into kernel language, and then we execute on the abstract machine
- This completes the proof