Welcome!

- Files and Programme at:
  `http://isabelle.in.tum.de/nominal/activities/cas09/`
- Have you already installed Isabelle?
- Can you step through Example.thy without getting an error message?

If yes, then very good.
If not, then please ask **now!**

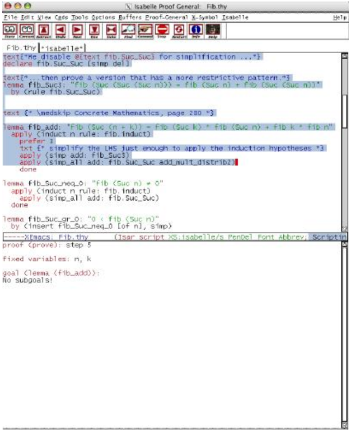
Nick Benton in "Machine Obstructed Proof":

> Automated proving is not just a slightly more fussy version of paper proving...It's a strange new skill, much harder to learn than a new programming language or application, or even many bits of mathematics...Coq is worth the bother and it, or something like it, is the future, if only we could make the initial learning experience a few thousand times less painful.

Same applies to Isabelle. So be prepared.

A Six-Slides Crash-Course on How to Use Isabelle

Proof General



Important buttons:

- Next and Undo advance / retract the processed part
- Goto jumps to the current cursor position, same as **ctrl-c/ctrl-return**

Feedback:

- warning messages are given in yellow
- error messages in red

X-Symbols

- …provide a nice way to input non-ascii characters; for example:

$$\forall\,,\ \exists\,,\ \Downarrow,\ \#,\ \bigwedge,\ \boldsymbol{\Gamma},\ \times,\ \neq,\ \in,\ \ldots$$

- they need to be input via the combination

  $\backslash$<name-of-x-symbol>

X-Symbols

- ...provide a nice way to input non-ascii characters; for example:

$$\forall\,, \exists\,, \Downarrow, \#, \bigwedge, \boldsymbol{\Gamma}, \times\,, \neq, \in, \ldots$$

- they need to be input via the combination

$$\backslash\textless\text{name-of-x-symbol}\textgreater$$

- short-cuts for often used symbols

$$\begin{array}{lll@{\qquad}rll@{\qquad}lll}
[| & \ldots & [\![& ==> & \ldots & \Longrightarrow & /\backslash & \ldots & \wedge \\
|] & \ldots &]\!] & => & \ldots & \Rightarrow & \backslash/ & \ldots & \vee
\end{array}$$

Isabelle Proof-Scripts

- Every proof-script (theory) is of the form

theory Name
  imports $T_1 \ldots T_n$
begin
...
end

Isabelle Proof-Scripts

- Every proof-script (theory) is of the form

```
theory Name
  imports T_1...T_n
begin
...
end
```

- Normally, one T will be the theory **Main**.

Types

- Isabelle is typed, has polymorphism and overloading.
  - Base types: nat, bool, string, ...
  - Type-formers: 'a list, 'a $\times$ 'b, 'c set, 'a $\Rightarrow$ 'b...
  - Type-variables: 'a, 'b, 'c, ...

Types

- Isabelle is typed, has polymorphism and overloading.
  - Base types: nat, bool, string, ...
  - Type-formers: 'a list, $'a \times 'b$, 'c set, $'a \Rightarrow 'b$...
  - Type-variables: 'a, 'b, 'c, ...

- Types can be queried in Isabelle using:

  typ nat
  typ bool
  typ string
  typ "('a $\times$ 'b)"
  typ "'c set"
  typ "'a list"
  typ "nat $\Rightarrow$ bool"

Terms

- The well-formedness of terms can be queried using:

```
term c
term "1::nat"
term 1
term "{1, 2, 3::nat}"
term "[1, 2, 3]"
term "(True, ''c'')"
term "Suc 0"
```

Terms

- The well-formedness of terms can be queried using:

  term c
  term "1::nat"
  term 1
  term "{1, 2, 3::nat}"
  term "[1, 2, 3]"
  term "(True, ''c'')"
  term "Suc 0"

- Isabelle provides some useful colour feedback

term "True"	gives	"True" :: "bool"
term "true"	gives	"true" :: "'a"
term "∀ x. P x"	gives	"∀ x. P x" :: "bool"

Formulae

- Every formula in Isabelle needs to be of type bool

```
term "True"
term "True ∧ False"
term "{1,2,3} = {3,2,1}"
term "∀ x. P x"
term "A ⟶ B"
```

Formulae

- Every formula in Isabelle needs to be of type bool

```
term "True"
term "True ∧ False"
term "{1,2,3} = {3,2,1}"
term "∀ x. P x"
term "A ⟶ B"
```

- When working with Isabelle, you are confronted with an objet logic (HOL) and a meta-logic (Pure)

```
term "A ⟶ B"    '='   term "A ⟹ B"
term "∀ x. P x"  '='   term "⋀x. P x"
```

Formulae

- Every formula in Isabelle needs to be of type bool

```
term "True"
term "True ∧ False"
term "{1,2,3} = {3,2,1}"
term "∀x. P x"
term "A ⟶ B"
```

- When working with Isabelle, you are confronted with an objet logic (HOL) and a meta-logic (Pure)

```
      term "A ⟶ B"   '='   term "A ⟹ B"
       term "∀x. P x"   '='   term "⋀x. P x"

term "A ⟹ B ⟹ C"   =   term "⟦A; B⟧ ⟹ C"
```

Inductive Predicates and Theorems

```
inductive
 even :: "nat ⇒ bool"
where
  eZ[intro]:  "even 0"
| eSS[intro]: "even n ⟹ even (Suc (Suc n))"
```

```
inductive
 even :: "nat ⇒ bool"
where
  eZ[intro]:  "even 0"
| eSS[intro]: "even n ⟹ even (Suc (Suc n))"
```

- The type of the predicate is always something to bool.
- The attribute [intro] adds the corresponding clause to the hint-theorem base (later more).
- The clauses correspond to the rules

$$\frac{}{\text{even } 0} \qquad \frac{\text{even } n}{\text{even } (\text{Suc } (\text{Suc } n))}$$

Theorems

- Isabelle's theorem database can be querried using

```
thm eZ
thm eSS
thm conjI
thm conjunct1
```

Theorems

- Isabelle's theorem database can be querried using

  thm eZ
  thm eSS
  thm conjI
  thm conjunct1

eZ:	even 0
eSS:	even ?n $\Longrightarrow$ even (Suc (Suc ?n))
conjI:	$[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$
conjunct1:	$?P \wedge ?Q \Longrightarrow ?P$

Theorems

- Isabelle's theorem database can be querried using

  thm eZ
  thm eSS
  thm conjI
  thm conjunct1

schematic variables

eZ:	even 0
eSS:	even ?n $\Longrightarrow$ even (Suc (Suc ?n))
conjI:	$[\![$?P; ?Q$]\!] \Longrightarrow$?P $\wedge$?Q
conjunct1:	?P $\wedge$?Q $\Longrightarrow$?P

Theorems

- Isabelle's theorem database can be querried using

```
thm eZ[no_vars]
thm eSS[no_vars]
thm conjI[no_vars]
thm conjunct1[no_vars]
```

attributes

eZ:	even 0
eSS:	$\text{even } n \Longrightarrow \text{even } (\text{Suc } (\text{Suc } n))$
conjI:	$[\![P;\, Q]\!] \Longrightarrow P \wedge Q$
conjunct1:	$P \wedge Q \Longrightarrow P$

Generated Theorems

- Most definitions result in automatically generated theorems; for example

```
thm even.intros[no_vars]
thm even.induct[no_vars]
```

Generated Theorems

- Most definitions result in automatically generated theorems; for example

```
thm even.intros[no_vars]
thm even.induct[no_vars]
```

intr's: even 0
even n $\Longrightarrow$ even (Suc (Suc n))

ind'ct: $[\![$even x; P 0;
$\bigwedge$n. $[\![$even n; P n$]\!]$ $\Longrightarrow$ P (Suc (Suc n))$]\!]$
$\Longrightarrow$P x

Theorem / Lemma / Corollary

- ... they are of the form:

```
theorem theorem_name:
  fixes     x::"type"
  ...
  assumes   "assm₁"
  and       "assm₂"
  ...
  shows   "statement"
  ...
```

- Grey parts are optional.
- Assumptions and the (goal)statement must be of type bool. Assumptions can have labels.

Theorem / Lemma / Corollary

- . . . they are

```
lemma even_double:
  shows "even (2 * n)"
...

lemma even_add:
  assumes a: "even n"
  and     b: "even m"
  shows "even (n + m)"
...

lemma neutral_element:
  fixes x::"nat"
  shows "x + 0 = x"
...
```

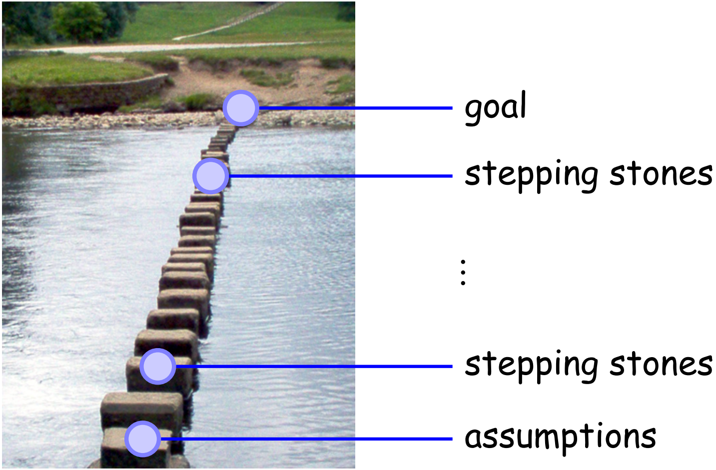
- Grey parts
- Assumptio … of type bool. Assumptions can have labels.

Isar Proofs about Even

An Isar Proof . . .

- The Isar proof language has been conceived by Markus Wenzel, the main developer behind Isabelle.

An Isar Proof . . .



- The Isar proof language has been conceived by Markus Wenzel, the main developer behind Isabelle.

An Isar Proof . . .

- A Rough Schema of an Isar Proof:

```
have    "assumption"
have    "assumption"
...
have    "statement"
have    "statement"
...
show "statement"
qed
```

An Isar Proof ...

- A Rough Schema of an Isar Proof:

```
have n1: "assumption"
have n2: "assumption"
...
have  n: "statement"
have  m: "statement"
...
show "statement"
qed
```

- each have-statement can be given a label

An Isar Proof . . .

- A Rough Schema of an Isar Proof:

```
have n1: "assumption" by justification
have n2: "assumption" by justification
...
have  n:  "statement" by justification
have  m:  "statement" by justification
...
show "statement" by justification
qed
```

- each have-statement can be given a label
- obviously, everything needs to have a justifiation

Justifications

- Omitting proofs

  sorry

- Assumptions

  by fact

- Automated proofs

by simp	simplification (equations, definitions)
by auto	simplification & proof search (many goals)
by force	simplification & proof search (first goal)
by blast	proof search

  ...

Justifications

- Omitting proofs

  sorry

- Assumptions

  by fact

- Automated proofs

  by simp

  by auto

  by force

  by blast

  ...

Automatic justifications can also be:

using ... by ...

using ih by ...

using n1 n2 n3 by ...

using lemma_name... by ...

First Exercise

- Lets try to prove a simple lemma. Remember we defined

> Eveness of a number:
>
> $$\frac{}{\text{even } 0}\,\text{eZ} \qquad \frac{\text{even } n}{\text{even } (\text{Suc } (\text{Suc } n))}\,\text{eSS}$$

```
lemma evan_double:
  shows "even (2 * n)"
```

First Exercise

- Lets try to prove a simple lemma. Remember we defined

Eveness of a number:

$$\frac{}{\text{even } 0}\,\text{eZ} \qquad \frac{\text{even } n}{\text{even } (\text{Suc } (\text{Suc } n))}\,\text{eSS}$$

```
lemma evan_double:
  shows "even (2 * n)"
proof (induct n)
```

Proofs by Induction

- Proofs by induction involve cases, which are of the form:

```
proof (induct)
 case (Case-Name x...)
  have "assumption" by justification
  ...
  have "statment" by justification
  ...
  show "statment" by justification
next
 case (Another-Case-Name y...)
  ...
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ}$$

$$\frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" sorry
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" sorry
  have a: "even (Suc (Suc (2 * n)))" sorry
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ} \qquad \frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" sorry
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" sorry
  have a: "even (Suc (Suc (2 * n)))" sorry
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ} \qquad \frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" sorry
  have a: "even (Suc (Suc (2 * n)))" sorry
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\ \text{eZ}$$

$$\frac{\text{even } n}{\text{even (Suc (Suc } n\text{))}}\ \text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" sorry
  have a: "even (Suc (Suc (2 * n)))" sorry
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\;\text{eZ} \qquad \frac{\text{even } n}{\text{even (Suc (Suc } n\text{))}}\;\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" by simp
  have a: "even (Suc (Suc (2 * n)))" sorry
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ}$$

$$\frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" by simp
  have a: "even (Suc (Suc (2 * n)))" sorry
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ} \qquad \frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" by simp
  have a: "even (Suc (Suc (2 * n)))" using ih by auto
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\ \text{eZ}$$

$$\frac{\text{even } n}{\text{even (Suc (Suc } n\text{))}}\ \text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" by simp
  have a: "even (Suc (Suc (2 * n)))" using ih by auto
  show "even (2 * (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ}$$

$$\frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_double:
  shows "even (2 * n)"
proof (induct n)
  case 0
  show "even (2 * 0)" by auto
next
  case (Suc n)
  have ih: "even (2 * n)" by fact
  have eq: "2 * (Suc n) = Suc (Suc (2 * n))" by simp
  have a: "even (Suc (Suc (2 * n)))" using ih by auto
  show "even (2 * (Suc n))" using eq a by simp
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,\text{eZ} \qquad \frac{\text{even } n}{\text{even (Suc (Suc } n))}\,\text{eSS}$$

```
lemma even_twice:
  shows "even (n + n)"
proof (induct n)
  case 0
  show "even (0 + 0)" sorry
next
  case (Suc n)
  have ih: "even (n + n)" by fact
  have eq: "(Suc n) + (Suc n) = Suc (Suc (n + n))" sorry
  have a: "even (Suc (Suc (n + n)))" sorry
  show "even ((Suc n) + (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\ eZ \qquad \frac{\text{even } n}{\text{even } (\text{Suc } (\text{Suc } n))}\ eSS$$

```
lemma even_twice:
  shows "even (n + n)"
proof (induct n)
  case 0
  show "even (0 + 0)" sorry
next
  case (Suc n)
  have ih: "even (n + n)" by fact
  have eq: "(Suc n) + (Suc n) = Suc (Suc (n + n))" sorry
  have a: "even (Suc (Suc (n + n)))" sorry
  show "even ((Suc n) + (Suc n))" sorry
qed
```

Your Turn

$$\frac{}{\text{even } 0}\ \text{eZ}$$

$$\frac{\text{even } n}{\text{even (Suc (Suc } n))}\ \text{eSS}$$

```
lemma even_twice:
  shows "even (n + n)"
proof (induct n)
  case 0
  show "even (0 + 0)" by auto
next
  case (Suc n)
  have ih: "even (n + n)" by fact
  have eq: "(Suc n) + (Suc n) = Suc (Suc (n + n))" by simp
  have a: "even (Suc (Suc (n + n)))" using ih by auto
  show "even ((Suc n) + (Suc n))" using eq a by simp
qed
```

Your Turn

$$\frac{}{\text{even } 0}\,eZ \qquad \frac{\text{even } n}{\text{even (Suc (Suc } n))}\,eSS$$

```
lemma even_twice:
  shows "even (n + n)"
proof (induct n)
  case 0
  show "even (0 + 0)" by auto
next
  case (Suc n)
  have ih: "even (n + n)" by fact
  have eq: "(Suc n) + (Suc n) = Suc (Suc (n + n))" by simp
  have "even (Suc (Suc (n + n)))" using ih by auto
  then show "even ((Suc n) + (Suc n))" using eq by simp
qed
```

A Chain of Facts

- Isar allows you to build a chain of facts as follows:

have n1: "..."
have n2: "..."

...

have ni: "..."
have "..." **using** n1 n2 ... ni

have "..."
moreover have "..."

...

moreover have "..."
ultimately have "..."

- also works for **show**

Your Turn

```
lemma even_twice:
  shows "even (n + n)"
proof (induct n)
  case 0
  show "even (0 + 0)" by auto
next
  case (Suc n)
  have ih: "even (n + n)" by fact
  have "(Suc n) + (Suc n) = Suc (Suc (n + n))" by simp
  moreover
  have "even (Suc (Suc (n + n)))" using ih by auto
  ultimately show "even ((Suc n) + (Suc n))" by simp
qed
```

Automatic Proofs

- Do not expect Isabelle to be able to solve automatically **show** "P=NP", but...

```
lemma
  shows "even (2 * n)"
by (induct n) (auto)

lemma
  shows "even (n + n)"
by (induct n) (auto)
```

Rule Inductions

Rule Inductions

- Remember we defined

Eveness of a number:

$$\frac{}{\text{even } 0}\,\mathsf{eZ} \qquad \frac{\text{even } n}{\text{even } (\text{Suc } (\text{Suc } n))}\,\mathsf{eSS}$$

Rule Inductions:

1.) Assume the property for the premises.
    Assume the side-conditions.

2.) Show the property for the conclusion.

Your Turn Again

```
lemma even_add:
  assumes a: "even n"
  and       b: "even m"
  shows "even (n + m)"
using a b
proof (induct)
  case eZ
  have as: "even m" by fact
  show "even (0 + m)"  sorry
next
  case (eSS n)
  have ih: "even m ⟹ even (n + m)" by fact
  have as: "even m" by fact

  show "even (Suc (Suc n) + m)" sorry
qed
```

Your Turn Again

```
lemma even_add:
  assumes a: "even n"
  and       b: "even m"
  shows "even (n + m)"
using a b
proof (induct)
  case eZ
  have "even m" by fact
  then show "even (0 + m)" by simp
next
  case (eSS n)
  have ih: "even m ⟹ even (n + m)" by fact
  have as: "even m" by fact
  have "even (n + m)" using ih as by simp
  then have "even (Suc (Suc (n + m)))" by auto
  then show "even (Suc (Suc n) + m)" by simp
qed
```

Rule Inductions

- Whenever a lemma is of the form

```
lemma
  assumes a: "pred"
  and     b: "somthing"
  shows "something_else"
```

with pred being an inductively defined predicate, then generally rule inductions are appropriate.

Does Not Work

```
lemma even_add_does_not_work:
  assumes a: "even n"
  and     b: "even m"
  shows "even (n + m)"
using a b
proof (induct n rule: nat_induct)
  case 0
  have "even m" by fact
  then show "even (0 + m)" by simp
next
  case (Suc n)
  have ih: "⟦even n; even m⟧ ⟹ even (n + m)" by fact
  have as1: "even (Suc n)" by fact
  have as2: "even m" by fact

  show "even ((Suc n) + m)"
```

Last Lemma about Even?

```
lemma even_mul:
  assumes a: "even n"
  shows "even (n * m)"
using a
proof (induct)
  case eZ
  show "even (0 * m)" by auto
next
  case (eSS n)
  have as: "even n" by fact
  have ih: "even (n * m)" by fact

  show "even ((Suc (Suc n)) * m)" sorry
qed
```

even_twice:	even (n + n)
even_add:	⟦even n; even m⟧ $\Longrightarrow$ even (n + m)

Last Lemma about Even?

```
lemma even_mul:
  assumes a: "even n"
  shows "even (n * m)"
using a
proof (induct)
  case eZ
  show "even (0 * m)" by auto
next
  case (eSS n)
  have as: "even n" by fact
  have ih: "even (n * m)" by fact

  show "even ((Suc (Suc n)) * m)" sorry
qed
```

even_twice: even (?n + ?n)

even_add: ⟦even ?n; even ?m⟧ ⟹ even (?n + ?m)

Last Lemma about Even?

```
lemma even_mul:
  assumes a: "even n"
  shows "even (n * m)"
using a
proof (induct)
  case eZ
  show "even (0 * m)" by auto
next
  case (eSS n)
  have ih: "even (n * m)" by fact
  have eq: "(m + m) + (n * m) = (Suc (Suc n)) * m" by simp
  have "even (m + m)" using even_twice by simp
  then have "even ((m + m) + (n * m))" using even_add ih by simp
  then show "even ((Suc (Suc n)) * m)" using eq by simp
qed
```

even_twice: even (n + n)

even_add: ⟦even n; even m⟧ $\Longrightarrow$ even (n + m)

Definitions

Definitions

- Often it is useful to define concepts in terms of existsing concepts. For example

```
definition
  divide :: "nat ⇒ nat ⇒ bool" ("_ DVD _" [100,100] 100)
where
  "m DVD n = (∃ k. n = m * k)"
```

- The annotation after the type introduces some more memorable syntax. The numbers are precedences.
- Once this definition is done, you can access it with

```
thm divide_def
m DVD n = (∃ k. n = m * k)
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃ k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃ k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

```
lemma even_divide:
  assumes a: "even n"
  shows "2 DVD n"
using a
proof (induct)
  case eZ
  have "0 = 2 * (0::nat)" by simp
  then show "2 DVD 0" by (auto simp add: divide_def)
next
  case (eSS n)
  have "2 DVD n" by fact
  then have "∃k. n = 2 * k" by (simp add: divide_def)
  then obtain k where eq: "n = 2 * k" by (auto)
  have "Suc (Suc n) = 2 * (Suc k)" using eq by simp
  then have "∃k. Suc (Suc n) = 2 * k" by blast
  then show "2 DVD (Suc (Suc n))" by (simp add: divide_def)
qed
```

Function Definitions and the Simplifier

Function Definitions

- Iterating a function n times can be defined by

```
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
| "f !! (Suc n) = (f !! n) o f"
```

Function Definitions

- Iterating a function n times can be defined by

a name

```
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
| "f !! (Suc n) = (f !! n) o f"
```

Function Definitions

- Iterating a function n times can be defined by

a type

```
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
| "f !! (Suc n) = (f !! n) o f"
```

Function Definitions

- Iterating a function n times can be defined by

pretty syntax

```
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
| "f !! (Suc n) = (f !! n) o f"
```

Function Definitions

- Iterating a function n times can be defined by

```
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
| "f !! (Suc n) = (f !! n) o f"
```

char. eqs

Function Definitions

- Iterating a function n times can be defined by

```
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
| "f !! (Suc n) = (f !! n) o f"
```

- Once a function is defined, the simplifier will be able to solve equations like

```
lemma
  shows "f !! (Suc (Suc 0)) = f o f"
  by (simp add: comp_def)
```

Your Turn

lemma shows "f !! (m + n) = (f !! m) o (f !! n)" **sorry**

A textbook proof: By induction on n:

- Case 0: Trivial.
- Case (Suc n): We have to show

$$f \,!!\, (m + (\mathrm{Suc}\ n)) = f \,!!\, m \circ (f \,!!\, (\mathrm{Suc}\ n))$$

The induction hypothesis is

$$f \,!!\, (m + n) = (f \,!!\, m) \circ (f \,!!\, n)$$

The justification

$$\begin{array}{lcll}
f \,!!\, (m + (\mathrm{Suc}\ n)) & = & f \,!!\, (\mathrm{Suc}\ (m + n)) & \\
 & = & f \,!!\, (m + n) \circ f & \\
 & = & (f \,!!\, m) \circ (f \,!!\, n) \circ f & \text{(by ih)} \\
 & = & (f \,!!\, m) \circ ((f \,!!\, n) \circ f) & \text{(by o_assoc)} \\
 & = & (f \,!!\, m) \circ (f \,!!\, (\mathrm{Suc}\ n)) &
\end{array}$$

Your Turn

```
lemma
  shows "f !! (m + n) = (f !! m) o (f !! n)"
proof (induct n)
  case 0
  show "f !! (m + 0) = (f !! m) o (f !! 0)" sorry
next
  case (Suc n)
  have ih: "f !! (m + n) = (f !! m) o (f !! n)" by fact

  show "f !! (m + (Suc n)) = f !! m o (f !! (Suc n))" sorry
qed
```

Your Turn

```
lemma
  shows "f !! (m + n) = (f !! m) o (f !! n)"
proof (induct n)
  case 0
  show "f !! (m + 0) = (f !! m) o (f !! 0)" by (simp add: comp_def)
next
  case (Suc n)
  have ih: "f !! (m + n) = (f !! m) o (f !! n)" by fact
  have eq1: "f !! (m + (Suc n)) = f !! (Suc (m + n))" by simp
  have eq2: "f !! (Suc (m + n)) = f !! (m + n) o f" by simp
  have eq3: "f !! (m + n) o f = (f !! m) o (f !! n) o f" using ih by simp
  have eq4: "(f !! m) o (f !! n) o f = (f !! m) o ((f !! n) o f)"
    by (simp add: o_assoc)
  have eq5: "(f !! m) o ((f !! n) o f) = (f !! m) o (f !! (Suc n))" by simp
  show "f !! (m + (Suc n)) = f !! m o (f !! (Suc n))"
    using eq1 eq2 eq3 eq4 eq5 by (simp only:)
qed
```

Equational Reasoning in Isar

- One frequently wants to prove an equation $t_1 = t_n$ by means of a chain of equations, like

$$t_1 = t_2 = t_3 = t_4 = \ldots = t_n$$

Equational Reasoning in Isar

- One frequently wants to prove an equation $t_1 = t_n$ by means of a chain of equations, like

$$t_1 = t_2 = t_3 = t_4 = \ldots = t_n$$

- This kind of reasoning is supported in Isar as:

```
have "t1 = t2" by just.
also have "... = t3" by just.
also have "... = t4" by just.
...
also have "... = tn" by just.
finally have "t1 = tn" by simp
```

Chains of Equations

```
lemma
  shows "f !! (m + n) = (f !! m) o (f !! n)"
proof (induct n)
  case 0
  show "f !! (m + 0) = (f !! m) o (f !! 0)" by (simp add: comp_def)
next
  case (Suc n)
  have ih: "f !! (m + n) = (f !! m) o (f !! n)" by fact
  have "f !! (m + (Suc n)) = f !! (Suc (m + n))" by simp
  also have "... = f !! (m + n) o f" by simp
  also have "... = (f !! m) o (f !! n) o f" using ih by simp
  also have "... = (f !! m) o ((f !! n) o f)" by (simp add: o_assoc)
  also have "... = (f !! m) o (f !! (Suc n))" by simp
  finally show "f !! (m + (Suc n)) = f !! m o (f !! (Suc n))" by simp
qed
```

Chains Involving Relations

- This type of reasoning also extends to relations.

```
fun
  pow :: "nat ⇒ nat ⇒ nat" ("_ ↑ _")
where
  "m ↑ 0 = 1"
| "m ↑ (Suc n) = m * (m ↑ n)"

lemma aux:
  fixes a b c::"nat"
  assumes a: "a ≤ b"
  shows " (c * a) ≤ (c * b)"
using a by (auto)
```

Chains Involving Relations

```
lemma
  shows "1 + n * x ≤ (1 + x) ↑ n"
proof (induct n)
  case 0
  show "1 + 0 * x ≤ (1 + x) ↑ 0" by simp
next
  case (Suc n)
  have ih: "1 + n * x ≤ (1 + x) ↑ n" by fact
  have "1 + (Suc n) * x ≤ 1 + x + (n * x) + (n * x * x)" by simp
  also have "... = (1 + x) * (1 + n * x)" by simp
  also have "... ≤ (1 + x) * ((1 + x) ↑ n)" using ih aux by blast
  also have "... = (1 + x) ↑ (Suc n)" by simp
  finally show "1 + (Suc n) * x ≤ (1 + x) ↑ (Suc n)" by simp
qed
```

Nested Proofs

```
lemma
  shows "n * x < (1 + x) ↑ n"
proof -
  have "1 + n * x ≤ (1 + x) ↑ n"
  proof (induct n)
    case 0
    show "1 + 0 * x ≤ (1 + x) ↑ 0" by simp
  next
    case (Suc n)
    have ih: "1 + n * x ≤ (1 + x) ↑ n" by fact
    have "1 + (Suc n) * x ≤ 1 + x + (n * x) + (n * x * x)" by (simp)
    also have "... = (1 + x) * (1 + n * x)" by simp
    also have "... ≤ (1 + x) * ((1 + x) ↑ n)" using ih aux by blast
    also have "... = (1 + x) ↑ (Suc n)" by simp
    finally show "1 + (Suc n) * x ≤ (1 + x) ↑ (Suc n)" by simp
  qed
  then show "n * x < (1 + x) ↑ n" by simp
qed
```

Isabelle Tutorial

I hope you want to do the whole proof about the compiler lemma for WHILE

- 9:00 - 11:00, Monday, 1 June
- 9:30 - 11:30, Tuesday, 2 June