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**!
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:

\[ \forall, \exists, \downarrow, \#, \land, \Gamma, \times, \neq, \in, \ldots \]

they need to be input via the combination

\[ \textbackslash<\text{name-of-x-symbol}> \]
X-Symbols

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

\[\forall, \exists, \downarrow, \#, \land, \Gamma, \times, \neq, \in, \ldots\]

they need to be input via the combination

\[\langle\text{name-of-x-symbol}\rangle\]

short-cuts for often used symbols

\[
\begin{bmatrix}
| & \ldots & | \\
| & \ldots & |
\end{bmatrix}
\Rightarrow \ldots \Rightarrow \left\langle \land \right\rangle \ldots \lor
\]

Beijing, 27. May 2009 - p. 5/49
Every proof-script (theory) is of the form

```isabelle
theory Name
  imports T_1 ... T_n
begin
...
end
```
Every proof-script (theory) is of the form

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

Normally, one $T$ will be the theory $\text{Main}$. 
Isabelle is typed, has polymorphism and overloading.

- **Base types**: `nat`, `bool`, `string`, ...
- **Type-formers**: `'a list`, `'a × 'b`, `'c set`, `'a ⇒ 'b`...
- **Type-variables**: `'a`, `'b`, `'c`, ...

Types can be queried in Isabelle using:
- `typ nat`
- `typ bool`
- `typ string`
- `typ ('a × 'b)`
- `typ 'c set`
- `typ 'a list`
- `typ nat ⇒ bool`
Types

Isabelle is typed, has polymorphism and overloading.

- Base types: nat, bool, string, ...
- Type-formers: 'a list, 'a × 'b, 'c set, 'a ⇒ 'b...
- Type-variables: 'a, 'b, 'c, ...

Types can be queried in Isabelle using:

```
typ nat
typ bool
typ string
typ "('a × 'b)"
typ "'c set"
typ "'a list"
typ "nat ⇒ bool"
```
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"`
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"
Every formula in Isabelle needs to be of type bool

- \texttt{"True"}
- \texttt{"True \land False"}
- \texttt{"\{1,2,3\} = \{3,2,1\}"}
- \texttt{"\forall x. P x"}
- \texttt{"A \rightarrow B"}
Every formula in Isabelle needs to be of type bool

- "True"
- "True ∧ False"
- "{1,2,3} = {3,2,1}"
- "∀ x. P x"
- "A ⟷ B"

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

- "A ⟷ B" = "A ⇒ B ⇒ C" = "[ [A; B] ] ⇒ C"
Formulae

- Every formula in Isabelle needs to be of type bool
  
  - "True"
  - "True ∧ False"
  - "{1,2,3} = {3,2,1}"
  - "∀ x. P x"
  - "A → B"

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

  - "A → B" = "A ⇒ B"
  - "∀ x. P x" = "⋀ x. P x"
  - "A → B → C" = "[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

  \[
  \begin{array}{c}
  \text{even } 0 \\
  \text{even } n \\
  \hline
  \text{even } 0 \\
  \text{even (Suc (Suc } n))
  \end{array}
  \]
Isabelle’s theorem database can be queried using:

- `thm eZ`
- `thm eSS`
- `thm conjI`
- `thm conjunct1`
Isabelle’s theorem database can be queried using:

- **thm eZ**
- **thm eSS**
- **thm conjI**
- **thm conjunct1**

**eZ:** even 0  
**eSS:** even ?n $\implies$ even (Suc (Suc ?n))  
**conjI:** \([?P; ?Q] \implies ?P \land ?Q\)  
**conjunct1:** \(?P \land ?Q \implies ?P\)
Theorems

Isabelle’s theorem database can be queried using:

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

**Example Theorems:****

- **eZ:** `even 0`
- **eSS:** `even ?n → even (Suc (Suc ?n))`
- **conjI:** `?P; ?Q → ?P ∧ ?Q`
- **conjunct1:** `?P ∧ ?Q → ?P`

**Schematic Variables:**

- `?n` represents a variable for natural numbers.
- `Suc` is the successor function.
- `even` is a predicate for even numbers.

Theorems

Isabelle’s theorem database can be queried using

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

Attributes:

- `eZ`: even 0
- `eSS`: even n \(\Rightarrow\) even (Suc (Suc n))
- `conjI`: \([P; Q] \Rightarrow P \land Q\)
- `conjunct1`: \(P \land Q \Rightarrow P\)
Most definitions result in automatically generated theorems; for example

\texttt{thm even.intros[no_vars]}
\texttt{thm even.induct[no_vars]}
Generated Theorems

- Most definitions result in automatically generated theorems; for example
  
  ```ml
  thm even.intros[no_vars]
  thm even.induct[no_vars]
  ```

 intra’s: `even 0`

- `even n → even (Suc (Suc n))`

 induce: `[even x; P 0; ∨ n. [even n; P n] → P (Suc (Suc n))]] → P x`
... they are of the form:

```plaintext
theorem theorem_name:
fixes x::"type"
...
assumes "assm_1"
and "assm_2"
...
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 of the form:

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

... Gray parts are optional.

```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 are not mandatory, they are often used to speed up the proof.

Assumptions and the (goal)statement must be of type bool. Assumptions can have labels.
Isar Proofs about Even
The Isar proof language has been conceived by Markus Wenzel, the main developer behind Isabelle.
The Isar proof language has been conceived by Markus Wenzel, the main developer behind Isabelle.
A Rough Schema of an Isar Proof:

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

Beijing, 27. May 2009 – p. 19/49
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 ...

Beijing, 27. May 2009 - p. 19/49
Let's try to prove a simple lemma. Remember we defined

Eveness of a number:

\[
\begin{align*}
\text{even } 0 & \quad \text{eZ} \\
\text{even } (\text{Suc } (\text{Suc } n)) & \quad \text{eSS}
\end{align*}
\]

Lemma `evan_double`:

shows "even (2 * n)"
First Exercise

Let's try to prove a simple lemma. Remember we defined

Eveness of a number:

\[
\begin{align*}
\text{eZ: even } 0 \\
\text{eSS: even } (\text{Suc } (\text{Suc } n)) \\
\end{align*}
\]

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 "statement" by justification
    ...
    show "statement" by justification
next
  case (Another-Case-Name y...)  
  ...
```
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
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
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
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
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
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
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
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
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
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


edZ
 even 0


eSS
 even n
e even (Suc (Suc n))
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
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

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
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
Do not expect Isabelle to be able to solve automatically show \( P=NP \), but...

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

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

- Remember we defined

Eveness of a number:

$$\begin{align*}
\text{even } 0 & \quad \text{even } (\text{Suc } (\text{Suc } n)) \\
\text{eZ} & \quad \text{eSS}
\end{align*}$$

Rule Inductions:

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

2.) Show the property for the conclusion.
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
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.
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] \implies even (n + m)" by fact
  have as1: "even (Suc n)" by fact
  have as2: "even m" by fact

  show "even ((Suc n) + m)"
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)
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
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] → even (n + m)
Definitions
Definitions

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

```haskell
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

```haskell
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
Iterating a function \( n \) times can be defined by

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

---

Function Definitions
Function Definitions

Iterating a function \( n \) times can be defined by

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

Beijing, 27. May 2009 - p. 40/49
Iterating a function $n$ times can be defined by

```haskell
fun
  iter :: "('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a)" ("_ !! _")
where
  "f !! 0 = (λx. x)"
  | "f !! (Suc n) = (f !! n) o f"
```
Iterating a function $n$ times can be defined by

\[
\text{fun}
\begin{align*}
\text{iter} & :: (\text{'a} \to \text{'a}) \Rightarrow \text{nat} \Rightarrow (\text{'a} \to \text{'a})" ("_ \text{!!} _") \\
\text{where} & \\
\text{"f} \text{!!} \text{0} & = (\lambda x. x) \\
\text{\mid } & \text{"f} \text{!!} (\text{Suc} \text{ n}) = (\text{f} \text{!! n}) \circ \text{f} 
\end{align*}
\]
Function Definitions

Iterating a function $n$ times can be defined by

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

char. eqs
Iterating a function \( n \) times can be defined by

\[
\text{fun iter :: } ('a \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow ('a \Rightarrow 'a)\text{"}_!!_"\text{"")}
\]

where

\[
\begin{align*}
\text{"f !! 0 = (\lambda x. x)"} \\
\text{"f !! (Suc n) = (f !! n) o f"}
\end{align*}
\]

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

\[
\text{lemma shows "f !! (Suc (Suc 0)) = f o f"}
\]

\[
\text{by (simp add: comp_def)}
\]
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

\[ f !! (m + (Suc n)) = f !! m o (f !! (Suc n)) \]

The induction hypothesis is

\[ f !! (m + n) = (f !! m) o (f !! n) \]

The justification

\[
\begin{align*}
f !! (m + (Suc n)) &= f !! (Suc (m + n)) \\
&= f !! (m + n) o f \\
&= (f !! m) o (f !! n) o f \quad \text{(by ih)} \\
&= (f !! m) o ((f !! n) o f) \quad \text{(by o_assoc)} \\
&= (f !! m) o (f !! (Suc n))
\end{align*}
\]
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 "\( f \circ (m + n) = (f \circ m) \circ (f \circ n) \)"

**Proof** (induct \( n \))

**Case** 0

show "\( f \circ (m + 0) = (f \circ m) \circ (f \circ 0) \)" by (simp add: comp_def)

**Next**

**Case** (Suc \( n \))

have ih: "\( f \circ (m + n) = (f \circ m) \circ (f \circ n) \)" by fact

have eq1: "\( f \circ (m + (Suc n)) = f \circ (Suc (m + n)) \)" by simp

have eq2: "\( f \circ (Suc (m + n)) = f \circ (m + n) \circ f \)" by simp

have eq3: "\( f \circ (m + n) \circ f = (f \circ m) \circ (f \circ n) \circ f \)" using ih by simp

have eq4: "\( (f \circ m) \circ (f \circ n) \circ f = (f \circ m) \circ ((f \circ n) \circ f) \)"

  by (simp add: o_assoc)

have eq5: "\( f \circ m \circ ((f \circ n) \circ f) = (f \circ m) \circ (f \circ (Suc n)) \)" by simp

show "\( f \circ (m + (Suc n)) = f \circ m \circ (f \circ (Suc n)) \)"

  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 "\( t_1 = t_2 \)" by just.
- also have "\( \ldots = t_3 \)" by just.
- also have "\( \ldots = t_4 \)" by just.
- \( \ldots \)
- also have "\( \ldots = t_n \)" by just.
- finally have "\( t_1 = t_n \)" 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
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)
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
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
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