# The Isar Proof Language

Simon Foster Jim Woodcock
University of York

16th August 2022

# Overview

1. Writing Properties and Proofs in Isar
2. Lemmas and Theorems
3. Equational Proofs with the Simplifier
4. Readable Proofs with Isar

# Outline

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"
```

```
definition doubleAll' :: "nat list ⇒ nat list" where
  "doubleAll' = map double"
```

- How do we show these functions are the same:

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"


definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same?

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases
- Formal proof allows us to show it holds for all cases

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"
```

```
definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same?

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases
- Formal proof allows us to show it holds for all cases

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"
```

```
definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same:

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"

definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same:

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"

definition doubleAll’ :: "nat list ⇒ nat list"
  where "doubleAll’ = map double"
```

- How do we show these functions are the same:

```
doubleAll = doubleAll’
```

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"

definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same:

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"

definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same:

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Motivation: Proof vs. Testing

- Consider two versions of the `doubleAll` function:

```
fun doubleAll :: "nat list ⇒ nat list" where
  "doubleAll [] = []" |
  "doubleAll (x # xs) = (x + x) # doubleAll xs"

definition doubleAll' :: "nat list ⇒ nat list"
  where "doubleAll' = map double"
```

- How do we show these functions are the same:

  `doubleAll = doubleAll'`

- We can test, but only for a finite number of cases.
- Formal proof allows us to show it holds for all cases.

# Outline

# Facts, Lemmas, and Theorems

- Commands like `datatype`, `definition`, and `fun` provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true; usually named.
- `print_theorems`: see facts generated by the previous command.
- `definition square :: "nat ⇒ nat" where "square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\ x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command `thm`.
- Named facts: created with commands `theorem` and `lemma`, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** square :: "nat ⇒ nat" **where** "square x = x*x"
- Produces fact square_def: definitional equation square x = x*x.
- $x$ is a free variable, and it can be instantiated with any value of type nat.
- Compare with $\lambda\, x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\ x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- `definition square :: "nat ⇒ nat" where "square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda\, x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** square :: "nat ⇒ nat" **where** "square x = x*x"
- Produces fact square_def: definitional equation square x = x*x.
- $x$ is a free variable, and it can be instantiated with any value of type nat.
- Compare with $\lambda\, x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda\, x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda\, x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\ x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- `definition square :: "nat ⇒ nat" where "square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Facts, Lemmas, and Theorems

- Commands like **datatype**, **definition**, and **fun** provide facts.
- These can all be used in a proof.
- Fact: a formula that the theorem prover accepts as true, usually named.
- **print_theorems**: see facts generated by the previous command.
- **definition** `square :: "nat ⇒ nat"` **where** `"square x = x*x"`
- Produces fact `square_def`: definitional equation `square x = x*x`.
- $x$ is a free variable, and it can be instantiated with any value of type `nat`.
- Compare with $\lambda x.\, x + y$, where $x$ is bound and $y$ is free.
- Recall the contents of a named theorem using the command **thm**.
- Named facts: created with commands **theorem** and **lemma**, and proofs.
- Lemma: smaller result, generally working towards a theorem.

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: Tn
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem name: "goal"`
- `fixes` give the free variables in a theorem, i.e. logical place holders
- `assumes` state any assumptions that the goal depends on.
- `shows` state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: T n
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem n: "goal"`.
- `fixes`: give the free variables in a theorem, i.e. logical place-holders.
- `assumes`: state any assumptions that the goal depends on.
- `shows`: state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: T n
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem n: "goal"`.
- `fixes`: give the free variables in a theorem, i.e. logical place-holders.
- `assumes`: state any assumptions that the goal depends on.
- `shows`: state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: T n
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem n: "goal"`.
- `fixes`: give the free variables in a theorem, i.e. logical place-holders.
- `assumes`: state any assumptions that the goal depends on.
- `shows`: state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: T n
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem n: "goal"`.
- `fixes`: give the free variables in a theorem, i.e. logical place-holders.
- `assumes`: state any assumptions that the goal depends on.
- `shows`: state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: T n
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem n: "goal"`.
- `fixes`: give the free variables in a theorem, i.e. logical place-holders.
- `assumes`: state any assumptions that the goal depends on.
- `shows`: state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# Specifying Theorems

- A theorem has this form:

```
theorem name:
  fixes x1 :: T1 ... xn :: T n
  assumes a1: "assm1" and a2: "assm2" ...
  shows "goal"
```

- Often a simpler form can be used, e.g. `theorem n: "goal"`.
- `fixes`: give the free variables in a theorem, i.e. logical place-holders.
- `assumes`: state any assumptions that the goal depends on.
- `shows`: state the goal that we want to prove.
- Example

```
theorem square_greater_zero:
  fixes x :: nat (* Type can be inferred. *)
  assumes "x > 0"
  shows "square x > 0"
```

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using by command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both $x$ and $y$ are free variables that can be instantiated with any value.
- "by" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using **by** command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both $x$ and $y$ are free variables that can be instantiated with any value.
- "**by**" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using by command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both $x$ and $y$ are free variables that can be instantiated with any value.
- "by" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using **by** command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both $x$ and $y$ are free variables that can be instantiated with any value.
- "**by**" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using **by** command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both $x$ and $y$ are free variables that can be instantiated with any value.
- "**by**" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using **by** command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both *x* and *y* are free variables that can be instantiated with any value.
- "by" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using **by** command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both $x$ and $y$ are free variables that can be instantiated with any value.
- "**by**" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# One-Line Proofs

- Equations and basic facts can be given to the simplifier.
- Increases automation of equational proofs.
- Often lets us prove a theorem in one line using **by** command.
- Example

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

- Both *x* and *y* are free variables that can be instantiated with any value.
- "**by**" takes a proof tactic that is applied to prove the theorem.
- If the tactic does not completely prove the goal, it fails.

# Outline

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be “simpler” than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be “simpler” than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

✓ $x + 0 = x$ ✓ $1 + 2 = 3$

✓ $x - x = 0$ ✗ $x + y = y + x$

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be “simpler” than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be "simpler" than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# The Simplifier

- Powerful proof tactic automating equational reduction of terms.
- Uses a form of fact called a simplification rule to rewrite the goal.
- Rules application repeated until no more simplifications are possible.

**Example**

| | |
|---|---|
| ✓ $x + 0 = x$ | ✓ $1 + 2 = 3$ |
| ✓ $x - x = 0$ | ✗ $x + y = y + x$ |

- LHS should be “simpler” than RHS (not enforced).
- Failure to ensure genuine simplification may lead to infinite rewrite loop.

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

### Theorem Attributes

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

**Theorem Attributes**

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

**Theorem Attributes**

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

**Theorem Attributes**

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

**Theorem Attributes**

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

## Theorem Attributes

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

## Theorem Attributes

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Applying the Simplifier

- Use `simp` or `(simp add: thms)` and `(simp only: thms)`.
- Mark theorems with attribute `[simp]` to make simplifier aware.
- theorem attributes allow us to provide hints to automated proof tactics.

## Theorem Attributes

```
(* Add a simplification rule, once proved *)
theorem mythm1 [simp]: "x + 0 = x" ...

(* Add proved rule as simplification *)
declare mythm2 [simp]

(* Remove simplification rule *)
declare mythm2 [simp del]
```

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

$$a + 0 = a \qquad (\texttt{add\_0\_right})$$
$$a + b = b + a \qquad (\texttt{add\_commute})$$
$$a * 0 = 0 \qquad (\texttt{mult\_0\_right})$$
$$a * (b + c) = a * b + a * c \qquad (\texttt{distrib\_left})$$
$$(a + b) * c = a * c + b * c \qquad (\texttt{distrib\_right})$$

- Command **`find_theorems`** searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

$$a + 0 = a \qquad (\texttt{add\_0\_right})$$
$$a + b = b + a \qquad (\texttt{add\_commute})$$
$$a * 0 = 0 \qquad (\texttt{mult\_0\_right})$$
$$a * (b + c) = a * b + a * c \qquad (\texttt{distrib\_left})$$
$$(a + b) * c = a * c + b * c \qquad (\texttt{distrib\_right})$$

- Command **`find_theorems`** searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

| | |
|---|---|
| $a + 0 = a$ | (`add_0_right`) |
| $a + b = b + a$ | (`add_commute`) |
| $a * 0 = 0$ | (`mult_0_right`) |
| $a * (b + c) = a * b + a * c$ | (`distrib_left`) |
| $(a + b) * c = a * c + b * c$ | (`distrib_right`) |

- Command **`find_theorems`** searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

| | |
|---|---|
| $a + 0 = a$ | (add_0_right) |
| $a + b = b + a$ | (add_commute) |
| $a * 0 = 0$ | (mult_0_right) |
| $a * (b + c) = a * b + a * c$ | (distrib_left) |
| $(a + b) * c = a * c + b * c$ | (distrib_right) |

- Command **find_theorems** searches for theorems matching pattern.
- "find_theorems "(+)"" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

| | |
|---|---|
| $a + 0 = a$ | (add_0_right) |
| $a + b = b + a$ | (add_commute) |
| $a * 0 = 0$ | (mult_0_right) |
| $a * (b + c) = a * b + a * c$ | (distrib_left) |
| $(a + b) * c = a * c + b * c$ | (distrib_right) |

- Command **find_theorems** searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

### Arithmetic Theorems

| | |
|---|---|
| $a + 0 = a$ | (`add_0_right`) |
| $a + b = b + a$ | (`add_commute`) |
| $a * 0 = 0$ | (`mult_0_right`) |
| $a * (b + c) = a * b + a * c$ | (`distrib_left`) |
| $(a + b) * c = a * c + b * c$ | (`distrib_right`) |

- Command **`find_theorems`** searches for theorems matching pattern.
- “`find_theorems "(+)"`” recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

## Arithmetic Theorems

| | |
|---|---|
| $a + 0 = a$ | (add_0_right) |
| $a + b = b + a$ | (add_commute) |
| $a * 0 = 0$ | (mult_0_right) |
| $a * (b + c) = a * b + a * c$ | (distrib_left) |
| $(a + b) * c = a * c + b * c$ | (distrib_right) |

- Command `find_theorems` searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

$$a + 0 = a \qquad (\texttt{add\_0\_right})$$
$$a + b = b + a \qquad (\texttt{add\_commute})$$
$$a * 0 = 0 \qquad (\texttt{mult\_0\_right})$$
$$a * (b + c) = a * b + a * c \qquad (\texttt{distrib\_left})$$
$$(a + b) * c = a * c + b * c \qquad (\texttt{distrib\_right})$$

- Command **find_theorems** searches for theorems matching pattern.
- "**find_theorems** "(+)"" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

| | |
|---|---|
| $a + 0 = a$ | (`add_0_right`) |
| $a + b = b + a$ | (`add_commute`) |
| $a * 0 = 0$ | (`mult_0_right`) |
| $a * (b + c) = a * b + a * c$ | (`distrib_left`) |
| $(a + b) * c = a * c + b * c$ | (`distrib_right`) |

- Command `find_theorems` searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

## Arithmetic Theorems

| | |
|---|---|
| $a + 0 = a$ | (`add_0_right`) |
| $a + b = b + a$ | (`add_commute`) |
| $a * 0 = 0$ | (`mult_0_right`) |
| $a * (b + c) = a * b + a * c$ | (`distrib_left`) |
| $(a + b) * c = a * c + b * c$ | (`distrib_right`) |

- Command **`find_theorems`** searches for theorems matching pattern.
- "`find_theorems "(+)"`" recalls all theorems containing plus operator.

# Theorem Library

- HOL contains a large library of arithmetic theorems.
- These help us to reason about the `square` function.

**Arithmetic Theorems**

$$a + 0 = a \qquad (\texttt{add\_0\_right})$$
$$a + b = b + a \qquad (\texttt{add\_commute})$$
$$a * 0 = 0 \qquad (\texttt{mult\_0\_right})$$
$$a * (b + c) = a * b + a * c \qquad (\texttt{distrib\_left})$$
$$(a + b) * c = a * c + b * c \qquad (\texttt{distrib\_right})$$

- Command **find_theorems** searches for theorems matching pattern.
- "**find_theorems** `"(+)"`" recalls all theorems containing plus operator.

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                    square_def
               =  x * (x + y) + y * (x + y)            distrib_right
               =  x * x + x * y + y * (x + y)          distrib_left
               =  x * x + x * y + (y * x + y * y)      distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                    square_def
               =  x * (x + y) + y * (x + y)            distrib_right
               =  x * x + x * y + y * (x + y)          distrib_left
               =  x * x + x * y + (y * x + y * y)      distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                  square_def
               =  x * (x + y) + y * (x + y)          distrib_right
               =  x * x + x * y + y * (x + y)        distrib_left
               =  x * x + x * y + (y * x + y * y)    distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                square_def
               =  x * (x + y) + y * (x + y)        distrib_right
               =  x * x + x * y + y * (x + y)      distrib_left
               =  x * x + x * y + (y * x + y * y)  distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) = (x + y) * (x + y)                   square_def
               = x * (x + y) + y * (x + y)           distrib_right
               = x * x + x * y + y * (x + y)         distrib_left
               = x * x + x * y + (y * x + y * y)     distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y = x * x + y * y + 2 * x * y
                            = x * x + y * y + (x + x) * y
                            = x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) = (x + y) * (x + y)                 square_def
               = x * (x + y) + y * (x + y)         distrib_right
               = x * x + x * y + y * (x + y)       distrib_left
               = x * x + x * y + (y * x + y * y)   distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y = x * x + y * y + 2 * x * y
                            = x * x + y * y + (x + x) * y
                            = x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                   square_def
               =  x * (x + y) + y * (x + y)           distrib_right
               =  x * x + x * y + y * (x + y)         distrib_left
               =  x * x + x * y + (y * x + y * y)     distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) = (x + y) * (x + y)               square_def
               = x * (x + y) + y * (x + y)       distrib_right
               = x * x + x * y + y * (x + y)     distrib_left
               = x * x + x * y + (y * x + y * y) distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y = x * x + y * y + 2 * x * y
                            = x * x + y * y + (x + x) * y
                            = x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                square_def
               =  x * (x + y) + y * (x + y)        distrib_right
               =  x * x + x * y + y * (x + y)      distrib_left
               =  x * x + x * y + (y * x + y * y)  distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y) =  (x + y) * (x + y)                square_def
               =  x * (x + y) + y * (x + y)        distrib_right
               =  x * x + x * y + y * (x + y)      distrib_left
               =  x * x + x * y + (y * x + y * y)  distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

```
theorem square_sum:
  "square (x + y) = square x + square y + 2*x*y"
  by (simp add: square_def algebra_simps)
```

## Left-hand side

```
square (x + y)  =  (x + y) * (x + y)                   square_def
                =  x * (x + y) + y * (x + y)           distrib_right
                =  x * x + x * y + y * (x + y)         distrib_left
                =  x * x + x * y + (y * x + y * y)     distrib_left
```

## Right-hand side

```
square x + square y + 2*x*y  =  x * x + y * y + 2 * x * y
                             =  x * x + y * y + (x + x) * y
                             =  x * x + y * y + x * y + x * y
```

# Step-by-step Proofs in Isar

- Single line proof with isn't always possible or desirable
- Isar provides a structured language for readable proofs
- This makes reasoning explicit
- Break down a proof into intermediate steps, combine to prove the goal
- Open a proof environment with delimiters
- : "quod erat demonstrandum": what was to be shown
- Intermediate facts proved using the command.
- Final fact proved using

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters `proof ... qed`.
- **qed**: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the **have** command.
- Final fact proved using **show**.

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters `proof ... qed`.
- `qed`: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the `have` command.
- Final fact proved using `show`.

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters `proof ... qed`.
- `qed`: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the `have` command.
- Final fact proved using `show`.

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters **proof ... qed**.
- **qed**: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the **have** command.
- Final fact proved using **show**.

# Step-by-step Proofs in Isar

- Single line proof with **`by`** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters **`proof`** `...` **`qed`**.
- **`qed`**: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the **`have`** command.
- Final fact proved using **`show`**.

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters **proof** ... **qed**.
- **qed**: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the **have** command.
- Final fact proved using **show**.

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters **proof** ... **qed**.
- **qed**: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the **have** command.
- Final fact proved using **show**.

# Step-by-step Proofs in Isar

- Single line proof with **by** isn't always possible or desirable.
- Isar provides a structured language for readable proofs.
- This makes reasoning explicit.
- Break down a proof into intermediate steps, combine to prove the goal.
- Open a proof environment with delimiters **proof** . . . **qed**.
- **qed**: "*quod erat demonstrandum*", what was to be shown.
- Intermediate facts proved using the **have** command.
- Final fact proved using **show**.

# Outline

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Example: Basic Proof in Isar

```
lemma square_calc:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have 1: "x + y = 1 + 2"
    by (simp add: assms) (* Use the assumptions *)
  from 1 have 2: "x + y = 3"
    by simp
  from 2 have 3: "square (x + y) = square 3"
    by simp
  from 3 have 4: "square (x + y) = 3 * 3"
    by (simp add: square_def)
  from 4 show "square (x + y) = 9"
    by simp
qed
```

# Proofs Commands

## Isar Proof Commands (Selection)

- `proof ... qed` delimiters for a proof block.
- `show "pred"` begin proof to demonstrate a subgoal.
- `have n: "pred"` begin proof of an intermediate fact.
- `by tactic` one line proof by application of `tactic`.
- `from n` bring an existing named fact into scope for a proof.
- `then` bring the previously proved fact into scope.
- `also` chain two equalities, $x = y$, $y = z \rightsquigarrow x = z$.
- `finally` final step in a chain of equality facts.
- `assume a: "pred"` introduce an assumption named a.
- `?thesis` schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- proof ... qed — delimiters for a proof block.
- show "pred" — begin proof to demonstrate a subgoal.
- have n: "pred" — begin proof of an intermediate fact.
- by "tactic" — one line proof by application of tactic.
- from n — bring an existing named fact into scope for a proof.
- then — bring the previously proved fact into scope.
- also — chain two equalities, $x = y, y = z \leadsto x = z$.
- finally — final step in a chain of equality facts.
- assume a: "pred" — introduce an assumption named a.
- ?thesis — schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- **proof** ... **qed** — delimiters for a proof block.
- **show** "pred" — begin proof to demonstrate a subgoal.
- **have** n: "pred" — begin proof of an intermediate fact.
- **by** tactic — one line proof by application of `tactic`.
- **from** n — bring an existing named fact into scope for a proof.
- **then** — bring the previously proved fact into scope.
- **also** — chain two equalities, $x = y$, $y = z \leadsto x = z$.
- **finally** — final step in a chain of equality facts.
- **assume** a: "pred" — introduce an assumption named a.
- ?thesis — schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- `proof ... qed` delimiters for a proof block.
- `show "pred"` begin proof to demonstrate a subgoal.
- `have n: "pred"` begin proof of an intermediate fact.
- `by tactic` one line proof by application of `tactic`.
- `from n` bring an existing named fact into scope for a proof.
- `then` bring the previously proved fact into scope.
- `also` chain two equalities, $x = y, y = z \leadsto x = z$.
- `finally` final step in a chain of equality facts.
- `assume a: "pred"` introduce an assumption named a.
- `?thesis` schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- `proof ... qed` delimiters for a proof block.
- `show "pred"` begin proof to demonstrate a subgoal.
- `have n: "pred"` begin proof of an intermediate fact.
- `by tactic` one line proof by application of `tactic`.
- `from n` bring an existing named fact into scope for a proof.
- `then` bring the previously proved fact into scope.
- `also` chain two equalities, $x = y$, $y = z \leadsto x = z$.
- `finally` final step in a chain of equality facts.
- `assume a: "pred"` introduce an assumption named a.
- `?thesis` schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- **proof** ... **qed** — delimiters for a proof block.
- **show** "pred" — begin proof to demonstrate a subgoal.
- **have** n: "pred" — begin proof of an intermediate fact.
- **by** tactic — one line proof by application of `tactic`.
- **from** n — bring an existing named fact into scope for a proof.
- **then** — bring the previously proved fact into scope.
- **also** — chain two equalities, $x = y$, $y = z \rightsquigarrow x = z$.
- **finally** — final step in a chain of equality facts.
- **assume** a: "pred" — introduce an assumption named a.
- ?thesis — schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- `proof ... qed` delimiters for a proof block.
- `show "pred"` begin proof to demonstrate a subgoal.
- `have n: "pred"` begin proof of an intermediate fact.
- `by tactic` one line proof by application of `tactic`.
- `from n` bring an existing named fact into scope for a proof.
- `then` bring the previously proved fact into scope.
- `also` chain two equalities, $x = y$, $y = z \rightsquigarrow x = z$.
- `finally` final step in a chain of equality facts.
- `assume a: "pred"` introduce an assumption named a.
- `?thesis` schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- `proof ... qed` delimiters for a proof block.
- `show "pred"` begin proof to demonstrate a subgoal.
- `have n: "pred"` begin proof of an intermediate fact.
- `by tactic` one line proof by application of `tactic`.
- `from n` bring an existing named fact into scope for a proof.
- `then` bring the previously proved fact into scope.
- `also` chain two equalities, $x = y$, $y = z \rightsquigarrow x = z$.
- `finally` final step in a chain of equality facts.
- `assume a: "pred"` introduce an assumption named a.
- `?thesis` schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- **proof** ... **qed** — delimiters for a proof block.
- **show** "pred" — begin proof to demonstrate a subgoal.
- **have** n: "pred" — begin proof of an intermediate fact.
- **by** tactic — one line proof by application of `tactic`.
- **from** n — bring an existing named fact into scope for a proof.
- **then** — bring the previously proved fact into scope.
- **also** — chain two equalities, $x = y$, $y = z \leadsto x = z$.
- **finally** — final step in a chain of equality facts.
- **assume** a: "pred" — introduce an assumption named a.
- ?thesis — schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- **proof** ... **qed** — delimiters for a proof block.
- **show** "pred" — begin proof to demonstrate a subgoal.
- **have** n: "pred" — begin proof of an intermediate fact.
- **by** tactic — one line proof by application of `tactic`.
- **from** n — bring an existing named fact into scope for a proof.
- **then** — bring the previously proved fact into scope.
- **also** — chain two equalities, $x = y$, $y = z \leadsto x = z$.
- **finally** — final step in a chain of equality facts.
- **assume** a: "pred" — introduce an assumption named a.
- ?thesis — schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- `proof ... qed` delimiters for a proof block.
- `show "pred"` begin proof to demonstrate a subgoal.
- `have n: "pred"` begin proof of an intermediate fact.
- `by tactic` one line proof by application of `tactic`.
- `from n` bring an existing named fact into scope for a proof.
- `then` bring the previously proved fact into scope.
- `also` chain two equalities, $x = y, y = z \leadsto x = z$.
- `finally` final step in a chain of equality facts.
- `assume a: "pred"` introduce an assumption named `a`.
- `?thesis` schematic variable for current goal predicate.

# Proofs Commands

## Isar Proof Commands (Selection)

- **proof** ... **qed** — delimiters for a proof block.
- **show** "pred" — begin proof to demonstrate a subgoal.
- **have** n: "pred" — begin proof of an intermediate fact.
- **by** tactic — one line proof by application of `tactic`.
- **from** n — bring an existing named fact into scope for a proof.
- **then** — bring the previously proved fact into scope.
- **also** — chain two equalities, $x = y$, $y = z \leadsto x = z$.
- **finally** — final step in a chain of equality facts.
- **assume** a: "pred" — introduce an assumption named `a`.
- ?thesis — schematic variable for current goal predicate.

# Example: Basic Proof in Isar (Alternative)

```
lemma square_calc_alt:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have "x + y = 1 + 2"
    by (simp add: assms)
  then have "x + y = 3"
    by simp
  hence "square (x + y) = square 3"
    by simp
  hence "square (x + y) = 3 * 3"
    by (simp add: square_def)
  then show ?thesis (* or use "thus ?thesis" *)
    by simp
qed
```

# Example: Basic Proof in Isar (Alternative)

```
lemma square_calc_alt:
  assumes "x = 1" "y = 2"
  shows "square (x + y) = 9"
proof -
  have "x + y = 1 + 2"
    by (simp add: assms)
  then have "x + y = 3"
    by simp
  hence "square (x + y) = square 3"
    by simp
  hence "square (x + y) = 3 * 3"
    by (simp add: square_def)
  then show ?thesis (* or use "thus ?thesis" *)
    by simp
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Example: Equational Proof in Isar

```
theorem square_sum:
  "square (x+y) = square x + square y + 2*x*y"
proof -
  have "square (x + y) = (x + y) * (x + y)"
    by (simp add: square_def)
  also have "... = (x + y) * x + (x + y) * y"
    by (simp add: distrib_left)
  also have "... = x * x + y * x + (x * y + y * y)"
    by (simp add: distrib_right)
  also have "... = x * x + y * y + x * y + x * y"
    by simp
  also have "... = square x + square y  + 2 * x * y"
    by (simp add: square_def)
  finally show ?thesis .
qed
```

# Conclusions

### This Lecture

- Definitions, theorems, and proofs.
- The simplifier.
- Readable proofs in Isar.

# Conclusions

### This Lecture

- Definitions, theorems, and proofs.
- The simplifier.
- Readable proofs in Isar.

# Conclusions

## This Lecture

- Definitions, theorems, and proofs.
- The simplifier.
- Readable proofs in Isar.

# Conclusions

## This Lecture

- Definitions, theorems, and proofs.
- The simplifier.
- Readable proofs in Isar.

# Conclusions

## This Lecture

- Definitions, theorems, and proofs.
- The simplifier.
- Readable proofs in Isar.