Isabelle Cheatsheet (2016-1)

定理・定義

⟨*theory*⟩ ::=

- **theorem** theorem_name:
  fixes x_1 :: "type 1" **and** ⋯ **and** x_ℓ :: "type ℓ"
  assumes ass_name_1: "ass 1" **and** ⋯ **and** ass_name_m: "ass m"
  shows sub_name_1: "claim 1" **and** ⋯ **and** sub_name_n: "claim n"
  ⟨*proof*⟩
  定理を言明．証明後は各 $i = 1 \ldots n$ について命題

  "$\bigwedge$x_1 ⋯ x_ℓ. ass 1 $\Longrightarrow$ ⋯ $\Longrightarrow$ ass m $\Longrightarrow$ claim i"

  (ただし，最外の $\bigwedge$ で束縛された変数 x_j は，常に schematic variable ?x_j に置き換えられる) が事実 (fact) となり，sub_name_i または theorem_name(i) と名前が付く．名前はどちらも省略可だが事実上どちらかは必要．

  証明のゴールは "claim 1" ⋯ "claim n" (唯一の場合 ?thesis でも表される)．証明中のみ "ass i" が fact となり ass_name_i または assms(i) と名前が付く．

  fixes は省略可．自由変数は $\bigwedge$ で束縛されているものとみなされる．

- 同義語：**lemma** = **proposition** = **corollary** = **theorem**

- **definition** f :: "type" **where** def_name: "f x_1 ⋯ x_n $\equiv$ right"
  非再帰的定義．定理の言明同様，自由変数 x_i は $\bigwedge$ で束縛されているものとみなされる．

  "$\bigwedge$x_1 ⋯ x_n. f x_1 ⋯ x_n $\equiv$ right"

  が fact となり，def_name (省略した場合は f_def) と名前が付き，[code] 属性が与えられる．

- **abbreviation** f :: "type" **where** "f x_1 ⋯ x_n $\equiv$ right"
  略記を定義．**definition** と違い定義を展開する必要がない．

- **fun** f :: "type"
  where name_1: "left_1 = right_1"
  | ⋯
  | name_n: "left_n = right_n"
  関数の再帰的定義．left_i は根記号 f のパターン．等式 "left_i = right_i" (の自由変数を $\bigwedge$ で束縛した命題) が fact となり，name_i または f.simps(i) と名前が付き，[simp, code] 属性が与えられる．マッチするパターンに関する場合わけ規則は f.cases，定義に関する帰納法スキームは f.induct．パターンの網羅性，停止性が自動で証明される．

- **fun** f_1 :: "type 1" **and** ⋯ **and** f_n :: "type n"
  where name_1: "left_1 = right_1"
  | ⋯
  | name_m: "left_m = right_m"
  相互再帰的関数定義．自動で生成される fact は f_1_⋯_f_n.simps 等となる．

- **function** f :: "type"
  where name_1: "left_1 = right_1"
  | ⋯
  | name_n: "left_n = right_n"
  ⟨*proof*⟩
  termination ⟨*proof*⟩
  ⋯ **fun** で自動証明ができない場合に使用．1 つ目の証明はパターン網羅性，2 つ目の証明は停止性．ほとんどの場合 1 つ目は pat_completeness メソッド，2 つ目は (relation "expression") メソッドで始める．"expression"は f の引数を比較する well-founded な二項関係（対の集合）を与える．

- **datatype** `('a1, ..., 'an) t =`
  `Const_1` ⟨*type*⟩* | ⋯ | `Const_`n ⟨*type*⟩*　再帰的データ型の宣言．場合分け規則 `t.cases` や，構造帰納法スキーム `t.induct` などの fact が作られる．

- **declare** ⟨*fact*⟩ [⟨*attr*⟩*]
  既知の fact の属性を変更．

補助コマンド

- **thm** ⟨*fact*⟩+ ⋯ fact を表示
- **find_theorems** ⟨*pattern*⟩ ⋯ パターンにマッチする fact を検索
- **print_theorems** ⋯ 直前のコマンドで作られた fact を表示

ロケール，クラス

- **context**
  `fixes x_1 :: "type 1" and ⋯ and x_`n `:: "type` n`"`
  `assumes name_1: "ass 1" and ⋯ and name_`m`: "ass` m`"`
  `begin` ⟨*theory*⟩* `end`　ブロック単位で仮定や変数を宣言．内部で `"claim"` として得た fact は，外部から参照すると

  $$\text{"}\bigwedge\texttt{x\_1}\ \cdots\ \texttt{x\_}n.\ \texttt{ass 1} \Longrightarrow \cdots \Longrightarrow \texttt{ass}\ m \Longrightarrow \texttt{claim"}$$

  となる．外部に見せたくない補題は `private` **lemma** などと記述する．

- **locale** `locale_name =` ⟨*locale expression*⟩ `+`
  `fixes x_1 :: "type 1" and ⋯ and x_`n `:: "type` n`"`
  `assumes name_1: "ass 1" and ⋯ and name_`m`: "ass` m`"`
  `begin` ⟨*theory*⟩* `end`　ロケールを定義．**context** に近いが，内部で `fact_name` として得た fact は外部から `locale_name.fact_name` で参照可．⟨*locale expression*⟩ で親ロケールを指定可．仮定が満たされる条件が述語 `locale_name` として定義される．

- **class** `class_name =` ⟨*class expression*⟩ `+`
  `fixes x_1 :: "type 1" and ⋯ and x_`n `:: "type` n`"`
  `assumes name_1: "ass 1" and ⋯ and name_`m`: "ass` m`"`
  `begin` ⟨*theory*⟩* `end`　クラスを定義．構文はロケールと近いが，型変数は `'a` のみが許される．内部で `fact_name` として得た fact は，`'a` にクラスが制限された形 (`'a :: class_name`) でグローバルに `fact_name` として参照可．

- **interpretation** `label: locale_name "arg 1" ⋯ "arg` n`"`
  `rewrites name_1: "l_1 = r_1" and ⋯ and name_`m`: "l_`m` = r_`m`"`
  ⟨*proof*⟩
  ロケールの前提を満たす引数を指定し，ロケールを解釈．証明のゴールは

  `"locale_name (arg 1) ⋯ (arg` n`)"`

  通常 `unfold_locales` メソッドを適用．証明後は `locale_name` 中 `name` で参照される fact や定数が `label.name` で参照可．名前の衝突がない場合 `label:` は省略化．

- **sublocale** `label: locale_name "arg 1" ⋯ "arg` n`"` ⟨*proof*⟩
  interpretation とほぼ同じだが，`end` 以降も関係が維持される．

- **subclass** `class_name` ⟨*proof*⟩
  現在のクラスが `class_name` のサブクラスであることを言明．

- **instance** `t :: (class_1, ···, class_n) class_name` ⟨*proof*⟩
  各型 `'a`i がクラス `class_`i に含まれるとき，データ型 `('a1, ···, 'an) t` がクラス `class_name` に含まれることを言明．証明のゴールは
  `"OFCLASS(('a1, ···, 'an) t, class_name)"`
  通常 `intro_classes` メソッドを適用．
- **instantiation** `t :: (class_1, ···, class_n) class_name`
  `begin`
  ⟨*theory*⟩*
  instance ⟨*proof*⟩
  `end`
  instance と同様だが定義が必要な場合に使用．
- **context** `name begin` ⟨*theory*⟩* `end`
  定義済みのロケール，クラスに言明を追加．
- 同義語：**theorem** `(in name) ···` =
  context `name begin` **theorem** `···` `end`
  同様の記法が **definition**，**subclass** などにも適用可．

補助コマンド

- **class_deps** ⟨*class*⟩? ⟨*class*⟩? ··· クラスの依存関係をグラフ化

証明

⟨*proof*⟩ ::=

- **by** ⟨*method*⟩
  証明メソッドを適用してゴールをすべて discharge．
- **apply** ⟨*method*⟩ ⟨*proof*⟩
  証明メソッドを適用してゴールを変形または discharge．
- **proof** ⟨*method*⟩ ⟨*isar*⟩* **qed** ⟨*method*⟩?
  まず 1 つ目の証明メソッドを適用 (何もしない場合 **proof**-)，残ったゴールを ⟨*isar*⟩* (後述) で discharge，さらに残ったゴールを最後の証明メソッドで discharge．

同義語

- `.` = **by** `assumption`
- `..` = **by** `standard`
- **done** = **by**-
- **using** ⟨*fact*⟩+ ≒ **apply** `(insert` ⟨*fact*⟩+`)`
- **unfolding** ⟨*fact*⟩+ ≒ **apply** `(unfold` ⟨*fact*⟩+`)`
  前者は書換え規則が 1 つも適用出来なくてもエラーにならないため (Isabelle 2016-1 現在)，保守性に難．
- **proof** ⟨*isar*⟩* = **proof** `standard` ⟨*isar*⟩*
- **by** ⟨*method*⟩ ⟨*method*⟩ = **proof** ⟨*method*⟩ **qed** ⟨*method*⟩
  内部的には **by** ⟨*method*⟩ は **proof** ⟨*method*⟩ **qed** であることによる．
  apply ⟨*method*⟩ **by** ⟨*method*⟩ や **by** (⟨*method*⟩, ⟨*method*⟩) を使えばよい．

Isar

⟨*isar*⟩ ::=

- **have** `name_1: "claim 1" and ⋯ and name_`n`: "claim `n`"` ⟨*proof*⟩
  `"claim `i`"` を証明し `name_`i という名前を付ける．
- **show** `name_1: "claim 1" and ⋯ and name_`n`: "claim `n`"` ⟨*proof*⟩
  have と同様だがゴール（のうち n 個）を証明して discharge.
- **note** `name [`⟨*attr*⟩$^+$`] =` ⟨*fact*⟩$^+$
  既知の fact を言明.
- **assume** `name_1: "ass 1" and ⋯ and name_`n`: "ass `n`"`
  `"ass `i`"` を仮定し `name_`i という名前を付ける．以下の形のゴールの証明に使用.

  $$\texttt{"ass 1} \Longrightarrow \cdots \texttt{ ass } n \Longrightarrow \texttt{ claim"}$$

- **fix** `x_1 :: "type 1" and ⋯ and x_`n` :: "type `n`"`
  `x_`i を，型 `"type `i`"` の任意の値とする．以下の形のゴールの証明に使用.

  $$\texttt{"}\bigwedge\texttt{x\_1 } \cdots \texttt{ x\_}n\texttt{. claim"}$$

- **define** `f :: "type" where name: "f x_1 ⋯ x_`n` ≡ right"`
  証明中でのみ有効な **definition**.
- **interpret** `locale_name "arg 1" ⋯ "arg `n`"` ⟨*proof*⟩
  証明中でのみ有効な **interpretation**.
- **obtain** `x_1 :: "type 1" and ⋯ and x_`n` :: "type `n`"`
  where `name_1: "claim 1" and ⋯ and name_`m`: "claim `m`"`
  ⟨*proof*⟩
  `"claim 1" ⋯ "claim `m`"`を満たす `x_1 ⋯ x_`n が存在することを証明し，以降の証明で利用.
- **case** `name: (Case_Name x_1 ⋯ x_`n`)`
  場合分け・帰納法中，`Case_Name` と名付けられたケースを考える．場合分けを表す仮定，帰納法の仮定が `name` (省略した場合 `Case_Name`) で参照可.
- **next**
  これまでに証明・仮定した fact などをすべてリセット．前提の異なるゴールの証明，別の場合分けに移る際に使用.
- `{` ⟨*isar*⟩$^+$ `}`
  ローカルスコープ．スコープ最後の言明に，スコープ内で **assume** した仮定を前提とし，**fix** した変数を $\bigwedge$ で束縛をした命題を **have** で示したのと同じ.
- **let** `?v = "expression"`
  メタ変数 `?v` を定義.
- ⟨*isar*⟩ **also** ⋯ **also** ⟨*isar*⟩ **finally** ⟨*isar*⟩
  推移性を持つ関係や等式を並べてひとつの結論にまとめる．直前の言明の右辺を表す `"..."` が便利．$>, \geq$ などは左右が逆になるので注意． 使用例：

  assume `"f x = a + b"`
  also have `"... ≤ c"` ⟨*proof*⟩
  also have `"... + d < e"` ⟨*proof*⟩
  finally have `"f x + d < e"`.

- ⟨*isar*⟩ **moreover** ⋯ **moreover** ⟨*isar*⟩ **ultimately** ⟨*isar*⟩
  複数の言明をまとめて最後の言明の前提に加える.

同義語

- **from** ⟨*fact*⟩$^+$ **have** ⋯ ⟨*proof*⟩ = **have** ⋯ **using** ⟨*fact*⟩$^+$ ⟨*proof*⟩
  ただし前者のみ，直前の言明を `this` として参照できる．
- **then** = **from** `this`
- **with** ⟨*fact*⟩$^+$ = **from** `this` ⟨*fact*⟩$^+$
- **hence** = **then have**
  使うメリットは少ない．
- **thus** = **then show**
  使うメリットは少ない．

証明メソッド

⟨*method*⟩ ::=

非自動的，堅牢

- `rule` ⟨*fact*⟩$^+$
  1 つ目のゴールの結論を fact（のうち，適用可能などれか）の結論として導く．適用された fact の前提が新たなゴールに加わる．
- `fact` ⟨*fact*⟩$^+$
  1 つ目のゴールを fact（のうち，適用可能などれか）から直接導いて discharge.
- `insert` ⟨*fact*⟩$^+$
  すべてのゴールの前提に既知の fact を加える．
- `assumption`
  1 つ目のゴールの結論を前提から直接導いて discharge.
- `intro` ⟨*fact*⟩$^+$
  適用できる限り `(rule` ⟨*fact*⟩$^+$`)` を適用（1 回も適用できなければ失敗）．
- `elim` ⟨*fact*⟩$^+$
  適用できる限り除去規則を適用（1 回も適用できなければ失敗）．
- `cases "expression"`
  `"expression"` のデータ構造に応じた場合分け．
- `cases "expression 1"` ⋯ `"expression` n`" rule:` ⟨*fact*⟩
  場合分け規則を指定した場合分け．
- `induct x arbitrary:` `y_`1 ⋯ `y_`n `rule:` ⟨*fact*⟩
  変数 `x` に関する帰納法を適用．帰納法の仮定において，`y_`1 ⋯ `y_`n は任意となる．`rule:`を省略すれば構造帰納法．
  帰納法内では元の `x` は無関係となるため，`x` に関する必要な fact は事前に `insert` で前提に含めてから `induct` を適用する．

- `subst (abs)`$^?$ (`n` $\cdots$) ⟨*fact*⟩
  現在のゴールに書換え規則を強制適用．与えられた fact は
  $$\texttt{"ass 1} \Longrightarrow \cdots \Longrightarrow \texttt{ass } m \Longrightarrow \texttt{left = right"}$$
  の形であり，ゴールの結論中 `left` にマッチする n 番目の式を `right` の形に書き換える．`"ass 1"` $\cdots$ `"ass` m`"` が新たなゴールに加わる．
- `unfold` ⟨*fact*⟩$^+$
  現在のゴールに 1 回以上，適用できる限り書換え規則を適用．
- `fold` ⟨*fact*⟩$^+$
  `unfold` の逆 (ただし，なぜかより保守的)．
- `atomize (full)`
  すべてのゴールを 1 つの HOL レベルの論理式にまとめる．複数のゴールに帰納法などを適用する場合に便利．

自動的

- `simp` {{`add` | `del` | `only`}`:` ⟨*fact*⟩$^+$}*
  現在のゴールに，`[simp]` 属性の与えられた fact を書換え規則として適用できる限り適用 (＋黒魔術)．パラメータ `add:` ⟨*fact*⟩$^+$, `del:` ⟨*fact*⟩$^+$ で書換え規則を制御可能．
- `simp_all` {{`add` | `del` | `only`}`:` ⟨*fact*⟩$^+$}*
  すべてのゴールに `simp` を適用．
- `auto` {{`simp` [`del`] | {`intro`[`!`] | `elim`[`!`] | `dest`[`!`] | `rule del`}`:` ⟨*fact*⟩$^+$}*
  すべてのゴールに登録済みの書換え規則，導入規則，除去規則（＋黒魔術）を適用．パラメータ `simp:` ⟨*fact*⟩$^+$, `intro:` ⟨*fact*⟩$^+$, 等で規則を制御可能．
- `force`, `fast`, `fastforce`, `blast`, `meson`, `metis`
  1 つ目のゴールを自動証明（証明できなければ失敗）．**sledgehammer** 等が見つけてきた場合のみ使い分ければよい．
- `linarith`, `arith`, `presburger`
  1 つ目のゴールに算術系に強い自動証明を適用．同上．
- `rule`
  現在のゴールに “標準” の導出規則を 1 回適用．メンテナンス上，適用される規則を明示する方がお勧め．
- `standard`
  現在のゴールに “標準” の証明メソッドを適用．同上．

合成

- (⟨*method*⟩)
  ⟨*method*⟩ が単語でない場合に **by** (⟨*method*⟩) などの形で必要．
- ⟨*method*⟩, ⟨*method*⟩
  ⟨*method*⟩ を順に適用．
- ⟨*method*⟩; ⟨*method*⟩
  1 つ目の ⟨*method*⟩ を適用して生成されたゴールすべてに 2 つ目の ⟨*method*⟩ を適用．
- ⟨*method*⟩`?` / ⟨*method*⟩`*` / ⟨*method*⟩`+`
  ⟨*method*⟩ を 1 回以下 / 0 回以上 / 1 回以上適用．何回適用する意図なのか分からないため，保守性が非常に悪い．

属性

name: "claim" の形の言明は name [⟨attr⟩+]: "claim" として属性を指定可.
⟨attr⟩ ::=

- simp [del]
  書換え規則. 証明メソッド simp や auto で使用される. 基本的に, 等式 "⋀x_1 ⋯ x_n. left = right" の形の fact に使用. 停止性に注意.

- intro[!|?]
  導入規則. 基本形は, my_pred を述語として,

  ```
  lemma my_predI [intro]:
    assumes "ass 1" and ⋯ and "ass n"
    shows "my_pred x_1 ⋯ x_m"
  ```

  x_i は変数でなくともよいが, 結論を出来るだけマッチしやすい形にするのが望ましい. 証明メソッド (intro my_predI) はゴールが

  $$\text{"}\cdots \Longrightarrow \texttt{my\_pred x\_1} \cdots \texttt{x\_}m\text{"}$$

  にマッチする場合に適用でき, 以下の n 個のゴールに置き換えられる.

  $$\begin{array}{c} \text{"}\cdots \Longrightarrow \texttt{ass 1}\text{"} \\ \vdots \\ \text{"}\cdots \Longrightarrow \texttt{ass } n\text{"} \end{array}$$

  intro! を指定した規則は auto 等で非可逆的に適用される.

- elim[!|?]
  除去規則. 基本形は, my_pred を述語として,

  ```
  lemma my_predE [elim]:
    assumes "my_pred x_1 ⋯ x_m"
      and "ass 1 1 ⟹ ⋯ ⟹ ass 1 n₁ ⟹ P"
      and ⋯
      and "ass ℓ 1 ⟹ ⋯ ⟹ ass ℓ nℓ ⟹ P"
    shows "P"
  ```

  ただし P は変数. x_i は変数でなくともよいが, 最初の前提を出来るだけマッチしやすい形にするのが望ましい. ! / ? については intro と同様.
  証明メソッド (elim my_predE) はゴールが

  $$\text{"}\cdots \Longrightarrow \texttt{my\_pred x\_1} \cdots \texttt{x\_}m \Longrightarrow \texttt{claim}\text{"}$$

  にマッチする場合に適用でき, 以下の ℓ 個のゴールに置き換わる.

  $$\begin{array}{c} \text{"}\cdots \Longrightarrow \texttt{ass 1 1} \Longrightarrow \cdots \Longrightarrow \texttt{ass 1 } n_1 \Longrightarrow \texttt{claim}\text{"} \\ \vdots \\ \text{"}\cdots \Longrightarrow \texttt{ass } \ell \texttt{ 1} \Longrightarrow \cdots \Longrightarrow \texttt{ass } \ell\ n_\ell \Longrightarrow \texttt{claim}\text{"} \end{array}$$

- dest[!|?]
  用途は elim と近く, ひとつしか仮定がない場合に以下の形で使用.

```
lemma my_predD [dest]:
  assumes "my_pred x_1 ··· x_m"
  shows "cond"
```

証明メソッド `(elim my_predD)` はゴールが

```
"··· ⟹ my_pred x_1 ··· x_m ⟹ claim"
```

にマッチする場合に適用でき，以下のゴールに置き換わる．

```
"··· ⟹ cond ⟹ claim"
```

- `trans`
  一般的な推移規則を宣言．たとえば

  ```
  "⋀a b c. a < b ⟹ b ≤ c ⟹ a < c"
  ```

  also による証明で利用される．
- `symm`
  関係の対称性を宣言．`"⋀a b. f a b ⟹ f b a"` の形の fact に指定する．`[symmetric]` 変形で利用される．
- `code`
  プログラムに変換される等式を指定する．仮定を持つコンテキストやロケール内で定義した関数は，コンテキスト外で `[code]` 宣言する必要がある (Isabelle 2016-1)．
- `code_unfold`
  プログラム変換を行う前処理として適用する書換え規則を指定．
- `unfolded` $\langle fact \rangle^+$
  参照中の fact に書換えを適用した結果を表す．
- `folded` $\langle fact \rangle^+$
  `unfolded` の逆．
- `simplified`
  参照中の fact に `[simp]` 属性の書換え規則を適用した結果を表す．
- `symmetric`
  `fact_name` の結論が等式のとき，`fact_name[symmetric]` は等式の左辺と右辺を交換したもの．
- `OF fact_1` ··· `fact_`n
  参照中の fact が `"ass 1 ⟹ ··· ass` n `⟹ claim"` の形のとき，各 `fact_`i が表す fact を `"ass` i`"` にマッチさせた結果の `"claim"` を表す．参照中の fact の形に依存するため，可変性に難が出る．
- `of "expr 1"` ··· `"expr` n`"`
  参照中の fact 中の schematic 変数を，順に `"expr` i`"` で置き換えた結果を表す．同上．

Isabelle/jEdit ショートカット

Ctrl-b	completion ヒント．tab で決定
Ctrl-e ↓	subscript
Ctrl-e ↑	superscript
`==>`	$\Longrightarrow$
`!!`	$\bigwedge$
`=>`	$\Rightarrow$
`-->`	$\longrightarrow$
`~`	$\neg$
`\noteq`	$\neq$
`>=`	$\geq$
`<=`	$\leq$
`->`	$\rightarrow$
`.>`	種々の左矢印
`<.`	種々の右矢印
`\forall`	$\forall$
`\exists`	$\exists$