Imperial College London

Department of Computing

Phometa — a proof assistant that build and prove a formal system based on visualisation

Author:
Gun Pinyo

Supervisor:
Dr. Krysia Broda

Second Marker:
Prof. Alessio R. Lomuscio

May 26, 2016

Submitted in part of fulfilment of the requirements
for the master of engineering

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Chapter 1

Introduction

1.1 Motivation

Proofs are very important to all kinds of Mathematics because they ensure the correctness of theorems. However, it is hard to verify the correctness of a proof itself especially for a complex proof. To tackle this problem, we can proof a theorem on *proof assistant*, aka *interactive theorem prover*, which does not allow us to make a mistake for each step of the proof, hence, when we finished writing a proof, we can make sure that it is correct.

There are many powerful and famous proof assistants such as Coq[9], Agda[4], and Isabelle[3] which are suitable for extreme use case of complex proofs. Nevertheless, they have steep learning curve and have specific meta-theory behind it, for example, Coq has Calculus of Inductive Construction (CIC), Agda has Unified Theory of Dependent Types[12][11] which are quite hard for newcomers. So I decided to create another proof assistants called *phometa* that is easy to use and require minimal prior knowledge.

To be precise, phometa is a tool to build a derivation tree. In order to achieve this, phometa has 3 kinds of nodes

- *Grammar* — How to construct a well-form term. For example, proposition can be either $\top \quad \bot \quad A \land B \quad A \lor B \quad \neg A \quad A \to B \quad A \leftrightarrow B$ where A and B are other arbitrary propositions.
- *Rule* (or derivation rule) — Reason that can be used to prove a term. For example, $A \land B$ is valid if we can prove that A is valid and B is valid.
- *Theorem* (or derivation tree) — An evidence (proof) that show that a term is valid.

By this 3 kinds of nodes. We can build any formal system (consists of grammars and rules) and prove its correctness as long as it support derivation tree.

Users can phometa by either

- Learn one of many existing formal systems provided in phometa's standard library and try to proof some theorem regarding to that formal system.
- Create their own formal system or extend an existing formal system, and do some experiments about it.

Once they get used to with this derivation tree builder. They can switch to more powerful proof assistants as mention above.

In order to make phometa easy to use, it is designed to be web-based application. Users will interact with phometa mainly by clicking buttons and pressing keyboard-shortcut. This has advantages over traditional proof assistant because it is easier to read, ill-from terms never occur, and guarantee that the entire system is always in consistent state.

1.2 Objectives

This project carries several objective as the following

- To make a construction of derivation-tree become more systematic. Hence, users become more productive and have less chance to make an error.
- To encourage users (especially students and newcomers) to create their own formal systems and reason about it.
- To integrate operational semantics with in formal system. Hence, be able represent the entire logical system with in just one framework.
- To show that most of formal systems have a similar meta-structure which can be implemented using common framework.
- To show advantages of visualised proof assistant over traditional one.

1.3 Achievement

- Finished designing phometa specification in such a way to keep it simple yet be able to produce a complex proof.
- Finished implementing phometa. All of basic functionality is working.
- Encoded Simple Arithmetic, Propositional Logic, and Typed Lambda Calculus as standard library in phometa.
- Write a tutorial for newcomers to use phometa (chapters 3, 4, 5, and 6).

Chapter 2

Related Work

There are many proof assistants available out there, each of them rely on slightly different meta-theory. We can separate proof assistants into 2 categories as the following

2.1 Text-Base Proof Assistants

Text-base proof assistants are similar to programming language where user writes everything in text-files and compile it, if the compilation is successful, then the proofs are correct. User can freely manipulate these text-files, hence, easier to write a complex proof. In addition, most of proof assistants have a plug-in to mainstream text editor, so user can use their favourite text editor with full performance.

There are several mainstream text-base proof assistants that worth mentioning

2.1.1 Coq

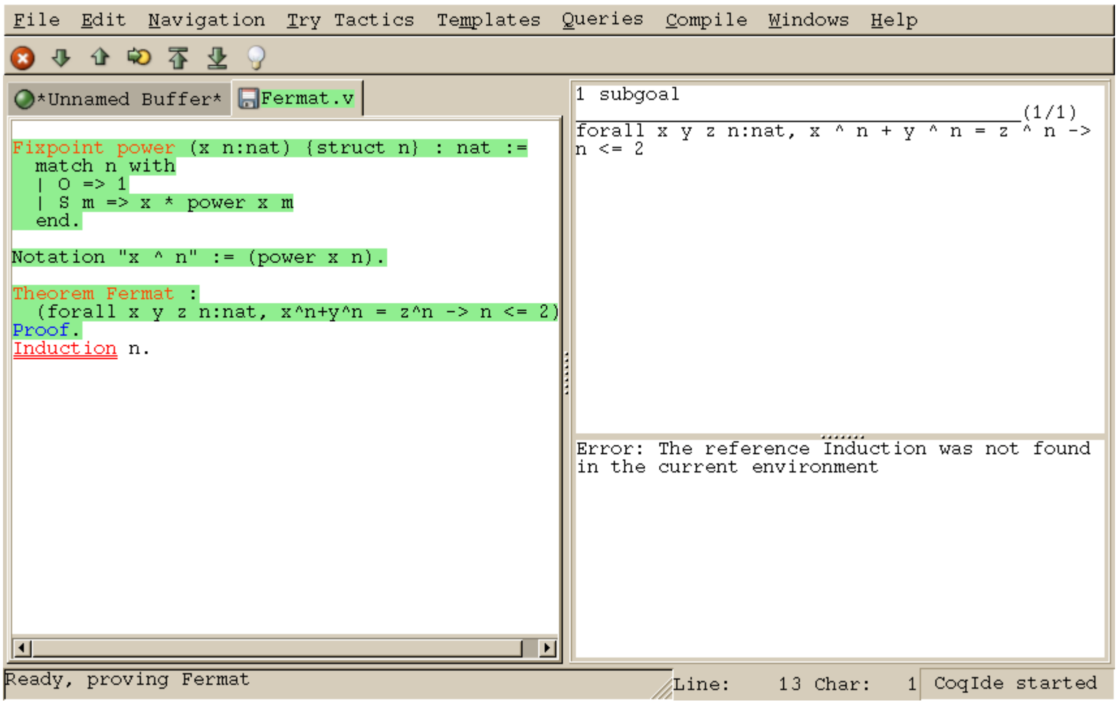
Coq[9] is one the most famous proof assistants. It is based on the Calculus of Inductive Constructions (CIC)[1] developed by Thierry Coquand[5].

Coq has customisable tactics which are commands that transform goal into smaller-sub goal (if any), this makes proving process become faster compared to other proof assistants. In contrast, tactics reduce readability, reader might need to replay each tactic step by step in order to understand a proof completely.

Coq is very mature, it has been developed since 1984. Hence, it is reliable and has lots of libraries supported.

[1] **CIC is itself is developed alongside Coq.**.

In term of editor, most people use Proof General[6] which is a plugin on Emacs[2]. Nevertheless, Coq has its own editor called CoqIde[10] that newcomers can use without learning Emacs.



```
Fixpoint power (x n:nat) {struct n} : nat :=
  match n with
  | O => 1
  | S m => x * power x m
  end.

Notation "x ^ n" := (power x n).

Theorem Fermat :
  (forall x y z n:nat, x^n+y^n = z^n -> n <= 2)
Proof.
Induction n.
```

```
1 subgoal
______________________________________(1/1)
forall x y z n:nat, x ^ n + y ^ n = z ^ n ->
n <= 2
```

```
Error: The reference Induction was not found
in the current environment
```

```
Ready, proving Fermat                Line:    13 Char:    1  CoqIde started
```

Figure 2.1: Screenshot of Coq (using CoqIde) — The left pane is file content and Upper right pane is the current goal which is changed depending on where the cursor point on file content.

2.1.2 Agda

Adga[4] is (dependently typed) functional programming which can be seen as a proof assistant as well. It is based on Unified Theory of Dependent Types[12][11] similar to Martin Lof Type Theory.

Its proving technique is relies on Curry-Howard correspondence which state that there is duality between computer programs and mathematical proofs[13], for example function corresponded to implication, product type corresponded to logical implication.

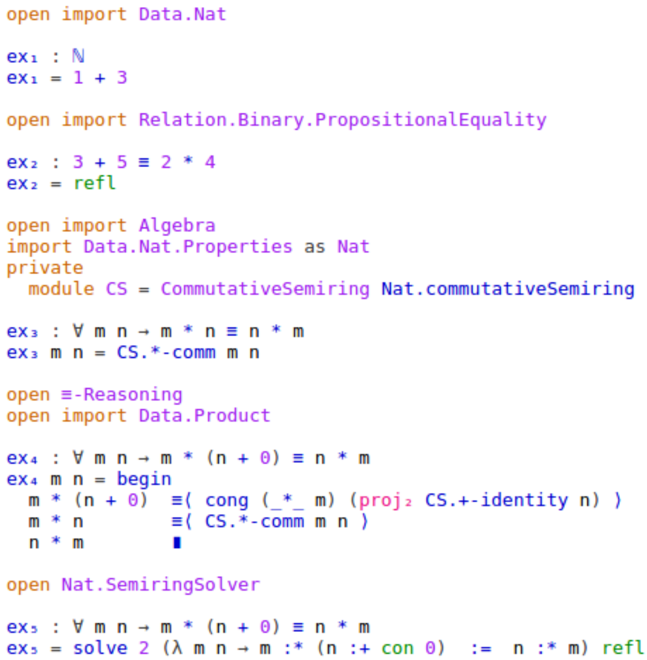
Agda is suitable for reasoning about functional programs because we can write a program and prove that curtain properties of a function hold using the same language. This is feasible since a proof is just a function due to Curry-Howard correspondence.

[2] Proof General also other proof assistants such as Isabelle and PhoX

Agda has less steep learning curve compared other proof assistants such as Coq. This is because user doesn't need to learn about proving system since it is the same as programming. In contrast, it doesn't have fancy tactic system so proving process is slower.

In term of popularity, it is less popular than Coq, however, some project such as Homotopy Type Theory[7][8] use Agda as alternative experiments to Coq.

In term of editor, Agda as its own plugin for Emacs which is very nice but user need to be familiar to Emacs before using it. There is no alternative plugin to other editor.



```
open import Data.Nat

ex₁ : ℕ
ex₁ = 1 + 3

open import Relation.Binary.PropositionalEquality

ex₂ : 3 + 5 ≡ 2 * 4
ex₂ = refl

open import Algebra
import Data.Nat.Properties as Nat
private
  module CS = CommutativeSemiring Nat.commutativeSemiring

ex₃ : ∀ m n → m * n ≡ n * m
ex₃ m n = CS.*-comm m n

open ≡-Reasoning
open import Data.Product

ex₄ : ∀ m n → m * (n + 0) ≡ n * m
ex₄ m n = begin
  m * (n + 0)  ≡⟨ cong (_*_ m) (proj₂ CS.+-identity n) ⟩
  m * n        ≡⟨ CS.*-comm m n ⟩
  n * m        ∎

open Nat.SemiringSolver

ex₅ : ∀ m n → m * (n + 0) ≡ n * m
ex₅ = solve 2 (λ m n → m :* (n :+ con 0)  :=  n :* m) refl
```

Figure 2.2: Screenshot of Agda — Credit: an example in Agda standard library, removing comment out to save space.

2.1.3 Isabelle

Isabelle[3] is generic proof assistant.

talk about isabelle readability

2.1.4 Lean

Lean**lean-offical-website** is a relatively new theorem prover[3]

[3] The Lean project was launched by Leonardo de Moura at Microsoft Research in 2013

2.2 Visualised Proof Assistant

TODO:

2.2.1 Logitext

http://logitext.mit.edu/tutorial

2.2.2 Panda

https://www.irit.fr/panda/

2.2.3 Pandora

http://www.doc.ic.ac.uk/pandora/newpandora/

2.2.4 PeaCoq

http://goto.ucsd.edu/peacoq/

2.2.5 Why3

http://why3.lri.fr/

Chapter 3

Background

In this chapter, we will go thought some required materials needed for later chapters. These can be linked together by an example of Simple Arithmetic explained below.

3.1 Formal System

A formal system is any well-defined system of abstract thought based on mathematical model[14]. Each formal system has a formal language composed of primitive symbols[1] acted by certain formation[2].

Informally, is an abstract system that has precise structures and can be reasoned about. For example, numbers (base 10) and their arithmetic (using + and ×) could form a formal system. This is because every term (e.g. 5, $(3+1)$, (3×4)) has explicit structure and we can argue something like "*does* 12 *equal to* (3×4)" or "*for any integers a and b, $(a+b)$ is equal to $(b+a)$*".

3.2 Backus-Naur Form

Backus-Naur Form (BNF) is a way to construct a term, for example, grammars of formal system above can be defined as the following

[1] phometa will assume that primitive symbols are any Unicode character.

```
<Digit>   ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

<Number> ::= <Digit> | <Number> <Digit>

<Expr>    ::= <Number>
            | ‘(’<Expr> ‘+’ <Expr>‘)’
            | ‘(’<Expr> ‘×’ <Expr>‘)’

<Equation> ::= <Expr> ‘=’ <Expr>
```

Figure 3.1: Backus-Naur Form of Simple Arithmetic

- A term of `<Digit>` can be either 0 or 1 or 2 or . . . or 9 and nothing else.

2 is `<Digit>` (3^{rd} choice)

Figure 3.2: This diagram explains that why 2 is a term of `<Digit>`

- A term of `<Number>` can be either
  - `<Digit>`
  - another `<Number>` concatenate with `<Digit>`

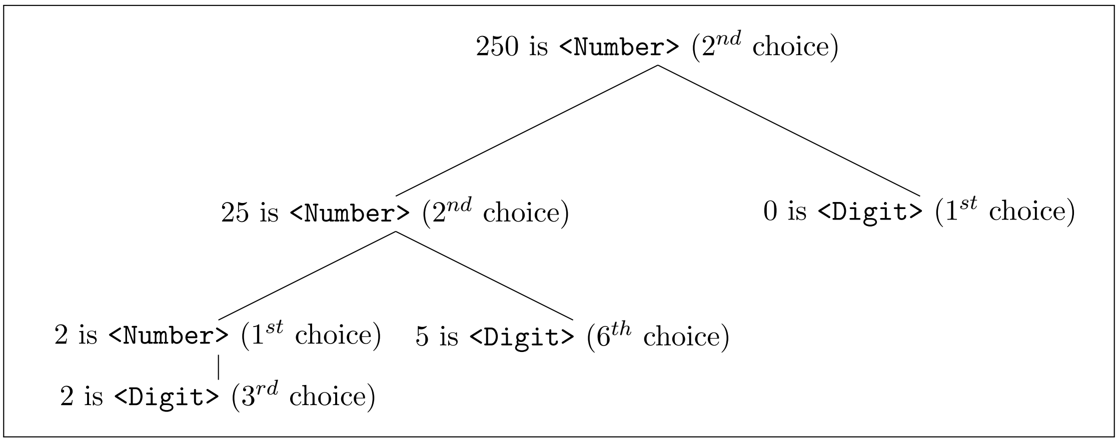

250 is <Number> (2nd choice)
25 is <Number> (2nd choice)
0 is <Digit> (1st choice)
2 is <Number> (1st choice)
5 is <Digit> (6th choice)
2 is <Digit> (3rd choice)

Figure 3.3: This diagram explains that why 250 is a term of `<Number>`

- A term of `<Expr>` can be either
  - `<Number>`
  - other two `<Expr>`s concatenate using '(' '+' ')'
  - other two `<Expr>`s concatenate using '(' '×' ')'

  Please note that we need brackets around '+' and '×' to avoid ambiguity. If we don't have these brackets, $3 + 4 + 5$ could be interpreted as either $(3 + 4) + 5$ or $3 + (4 + 5)$ which is not precise. Moreover, $12 + 0 \times 6$ will be interpreted as $12 + (0 \times 6)$ due to priority of × over + and it is impossible to encode some thing like $(12 + 0) \times 6$.

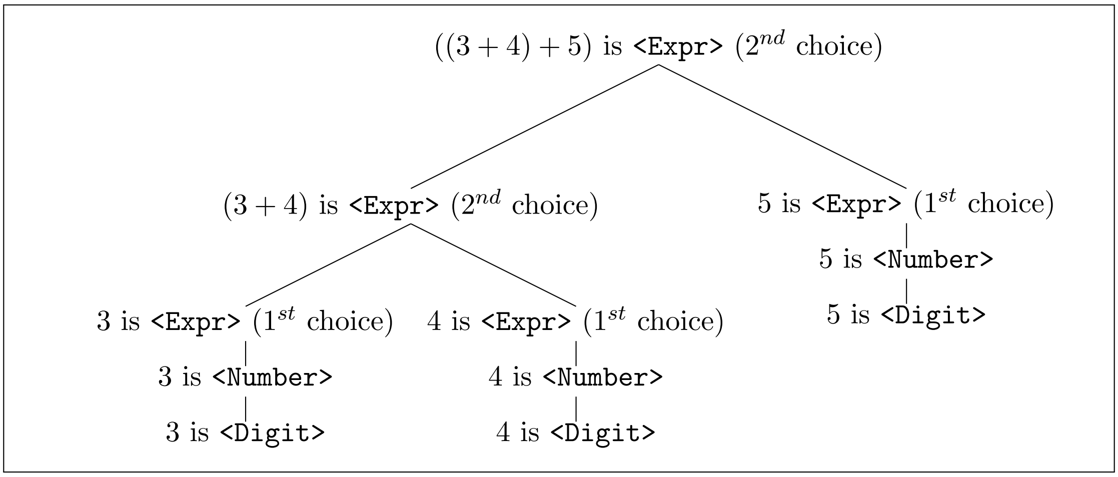


Figure 3.4: This diagram explains that why $((3 + 4) + 5)$ is a term of `<Expr>`

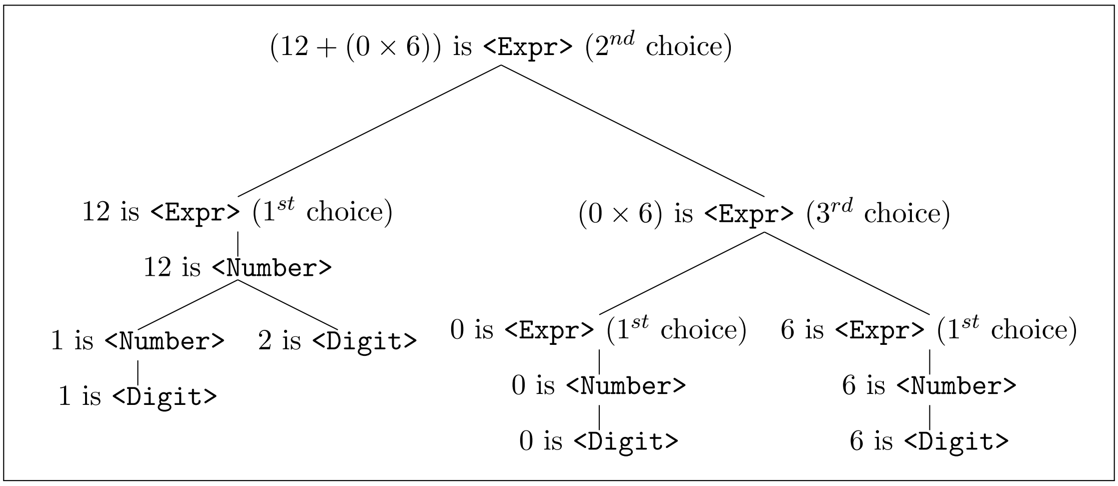


Figure 3.5: This diagram explains that why $(12 + (0 \times 6))$ is a term of `<Expr>`

- A term of <Equation> can be only two <Expr>s concatenate using '='

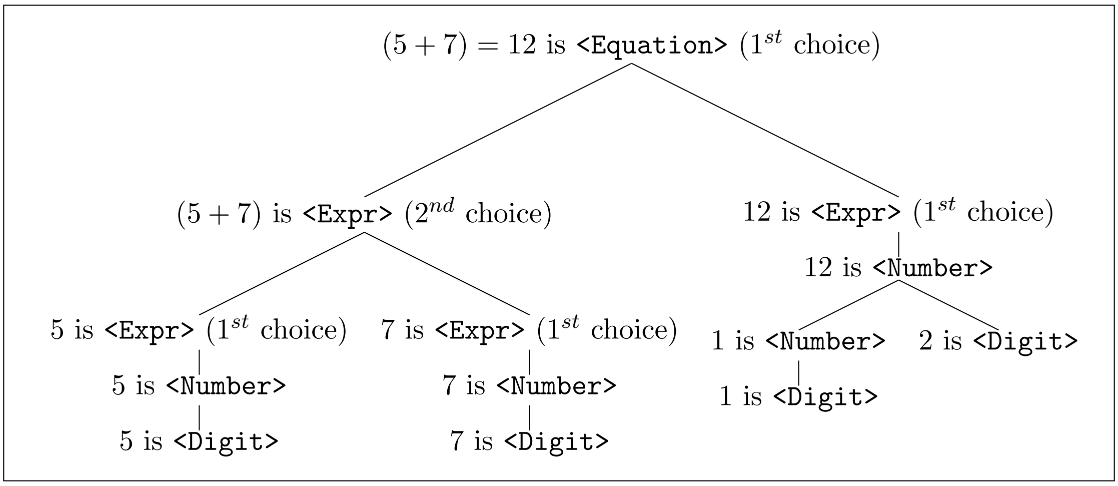

$(5+7)=12$ is <Equation> (1^{st} choice)
$(5+7)$ is <Expr> (2^{nd} choice)
12 is <Expr> (1^{st} choice)
12 is <Number>
5 is <Expr> (1^{st} choice)
7 is <Expr> (1^{st} choice)
1 is <Number>
2 is <Digit>
5 is <Number>
7 is <Number>
1 is <Digit>
5 is <Digit>
7 is <Digit>

Figure 3.6: This diagram explains that why $(5+7)=12$ is a term of <Equation>

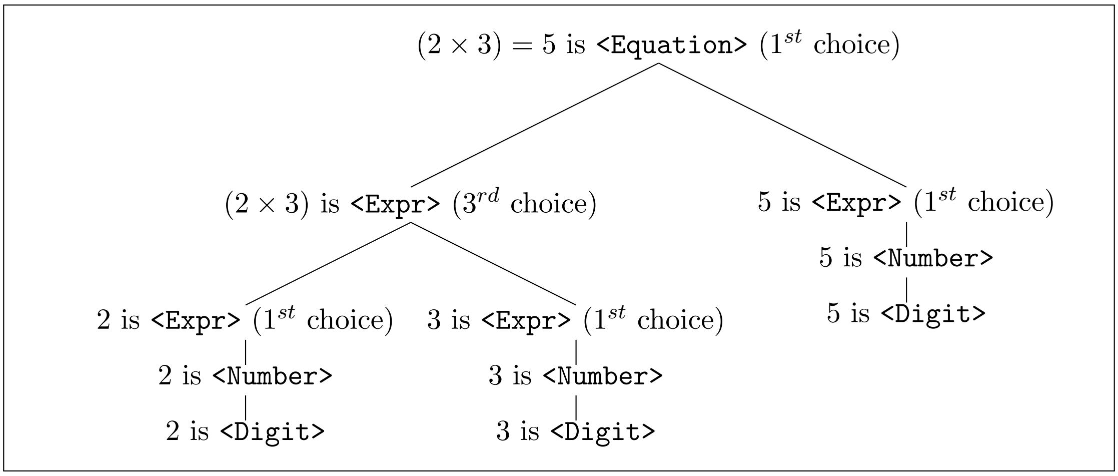

$(2\times 3)=5$ is <Equation> (1^{st} choice)
(2×3) is <Expr> (3^{rd} choice)
5 is <Expr> (1^{st} choice)
5 is <Number>
2 is <Expr> (1^{st} choice)
3 is <Expr> (1^{st} choice)
5 is <Digit>
2 is <Number>
3 is <Number>
2 is <Digit>
3 is <Digit>

Figure 3.7: This diagram explains that why $(2\times 3)=5$ is a term of <Equation>. Please note that this construction is purely syntactic so wrong equation is acceptable.

3.3 Meta Variables and Pattern Matching

Meta variables are arbitrary sub-terms embedded inside root term. For example, an <Expr> $(x+y)$ represents two arbitrary <Expr> joined by '+'.

But if we have an `<Equation>` $(x + 7) = 12$, shouldn't x be an unknown variable that needed to be solve rather than being arbitrary `<Expr>`? Well, x still represents arbitrary `<Expr>` but in order make this equation hold, x must be 5. Hence "*variable needed to be solve*" is just spacial form of "*variable as arbitrary term*".

Meta variables help us to represents statement in more general manner. For example, "*the same expressions plus together is the same as 2 times that expression*" could be represented by $(x + x) = (2 \times x)$ rather than $(0 + 0) = (2 \times 0)$ and $(1 + 1) = (2 \times 1)$ and $(2 + 2) = (2 \times 2)$ and so on.

But if we know that $(x+x) = (2\times x)$, how could we derive its instance e.g. $(1+1) = (2\times 1)$ or even $(y \times z) + (y \times z) = (2 \times (y \times z))$? The solution for this is to use *Pattern Matching* which is algorithm that try to substitute pattern's meta variables into more specific form, in order to make pattern identical to target, for example

- $(x + x) = (2 \times x)$ is pattern matchable with $(1 + 1) = (2 \times 1)$ by substitute x with1
- $(x + x) = (2 \times x)$ is pattern matchable with $(y \times z) + (y \times z) = (2 \times (y \times z))$ by substitute x with $(y \times z)$
- $(x + x) = (2 \times x)$ is *not* pattern matchable with $(1 + 1) = 2$, if we try to substitute x with 1 we would get $(1 + 1) = (2 \times 1)$ which is not identical to $(1 + 1) = 2$
- $(1 + 1) = (2 \times 1)$ is *not* pattern matchable with $(x + x) = (2 \times x)$, because pattern $(1 + 1) = (2 \times 1)$ doesn't have any meta variable and it is not identical to $(x + x) = (2 \times x)$. This show that pattern matching doesn't generally holds in opposite direction
- $(x + x) = (2 \times x)$ is pattern matchable to itself by substitute x with x

If pattern matching is successful then the target is instance of the target is instance of the pattern.

3.4 Derivation of Formal Systems

So far, we construct any term based on Backus-Naur Form, this doesn't prevent invalid term, for example, $(2 \times 3) = 5$ is perfectly a term of `<Equation>`. Thus, we need some mechanism to verify a term i.e. *prove* that the particular term holds. One way to deal with this is to use derivation system, first, we have a set of derivation rules that has format as the following

$$\textsc{rule-name}\ \frac{Premise_1 \qquad Premise_2 \qquad Premise_3 \qquad \ldots \qquad Premise_n}{Conclusion}$$

Figure 3.8: Structure of derivation rule.

This say that if we know that $Premise_1$ and $Premise_2$ and $Premise_3$ and $\ldots$ and $Premise_n$ hold then $Conclusion$ holds. In another word, if we want to prove $Conclusion$ then we can use this derivation rule then proof its premises.

Derivation rules of current example formal system could be shown as the following

$$\textsc{eq-refl}\ \frac{}{x = x} \qquad \textsc{eq-symm}\ \frac{y = x}{x = y} \qquad \textsc{eq-tran}\ \frac{x = z \qquad z = y}{x = y}$$

$$\textsc{add-intro}\ \frac{u = x \qquad v = y}{(u + v) = (x + y)} \qquad \textsc{mult-intro}\ \frac{u = x \qquad v = y}{(u \times v) = (x \times y)}$$

$$\textsc{add-assoc}\ \frac{}{((x + y) + z) = (x + (y + z))} \qquad \textsc{mult-assoc}\ \frac{}{((x \times y) \times z) = (x \times (y \times z))}$$

$$\textsc{add-comm}\ \frac{}{(x + y) = (y + x)} \qquad \textsc{mult-comm}\ \frac{}{(x \times y) = (y \times x)}$$

$$\textsc{dist-left}\ \frac{}{(x \times (y + z)) = ((x \times y) + (x \times z))} \qquad \textsc{dist-right}\ \frac{}{((x + y) \times z) = ((x \times z) + (y \times z))}$$

Figure 3.9: Derivation rules of Simple Arithmetic (not exhaustive, due to limited space).

In order to use a derivation rule, first the conclusion of the rule is pattern match against current goal, if it is pattern matchable then meta variables in premises are substituted respect to the pattern matching (if some meta variables of premises doesn't exist in substitution list then we are free to substitute by anything). These substituted premises will become next goals that we need to prove.

For example if we want to prove $((3+4)*5) = ((4+3)*5)$ we could use rule MULT-INTRO to prove it since $(u \times v) = (x \times y)$ is pattern matchable with $((3+4)*5) = ((4+3)*5)$ by substitute u with $(3+4)$, v with 5, x with $(4+3)$, and y with 5. Then premises $u = x$ and $v = y$ are substituted and become $(3+4) = (4+3)$ and $5 = 5$ respectively. Therefore, $((3+4)*5) = ((4+3)*5)$ can be proven by MULT-INTRO and produce another two sub-goals which are $(3+4) = (4+3)$ and $5 = 5$. This can be shown as instance of MULT-INTRO as the following

$$\dfrac{(3+4)=(4+3) \qquad 5=5}{((3+4)*5)=((4+3)*5)}\ \textsc{add-intro}$$

Figure 3.10: Example of instance of derivation rule.

For the remaining, we could prove $(3+4)=(4+3)$ using ADD-COMM because $(x+y)=(y+x)$ is pattern matchable with $(3+4)=(4+3)$, ADD-COMM doesn't have any premises hence there are no further sub-goal. For $5=5$ we could use EQ-REFL, this also deosn't produce further sub-goal so the entire proof is complete. We can the write the entire proof using *derivation tree* as the following

$$\dfrac{\dfrac{}{(3+4)=(4+3)}\ \textsc{add-comm} \qquad \dfrac{}{5=5}\ \textsc{eq-refl}}{((3+4)*5)=((4+3)*5)}\ \textsc{add-intro}$$

Figure 3.11: Example of derivation tree.

Some rules in figure 3.9 don't have any premises. This is necessary, otherwise, applying rule always generate further sub goals and never terminate. These rules can be seen as *axiom* which is a term that valid by assumption i.e. so need to prove such a term.

For better understanding about derivation system, here is a more complex derivation tree which prove $(((w\times x)+(w\times y))\times z)=(w\times((x\times z)+(y\times z)))$. Reader is encouraged to explore that why this derivation tree is correct.

$$\dfrac{\begin{array}{c}\dfrac{}{((w\times(x+y))\times z)=(w\times((x+y)\times z))}\ \textsc{mult-assoc}\\ \vdots\end{array} \qquad \dfrac{\dfrac{}{w=w}\ \textsc{eq-refl} \qquad \dfrac{}{((x+y)\times z)=((x\times z)+(y\times z))}\ \textsc{dist-right}}{(w\times((x+y)\times z))=(w\times((x\times z)+(y\times z)))}\ \textsc{mult-intro}}{((w\times(x+y))\times z)=(w\times((x\times z)+(y\times z)))}\ \textsc{eq-tran}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(w\times(x+y))=((w\times x)+(w\times y))}\ \textsc{dist-left}}{((w\times x)+(w\times y))=(w\times(x+y))}\ \textsc{eq-symm} \qquad \dfrac{}{z=z}\ \textsc{eq-refl}}{(((w\times x)+(w\times y))\times z)=((w\times(x+y))\times z)}\ \textsc{mult-intro} \qquad \vdots}{(((w\times x)+(w\times y))\times z)=(w\times((x\times z)+(y\times z)))}\ \textsc{eq-tran}$$

Figure 3.12: Example of more complex derivation tree.

Chapter 4

Example Formal System — Simple Arithmetic

As in background chapter, Simple Arithmetic is used as example to explain basic concept of formal systems and its derivations. In order to make the transition goes smoother, this chapter aims to encode Simple Arithmetic and explain basic features and usability of *phometa* at the same time. Please note that this is just a faction of actual arithmetic modified to make it easier to understand, so it is not as powerful as the actual one.

4.1 First time with phometa

You can download complied version of phometa at

```
https://github.com/gunpinyo/phometa/raw/master/build/phometa.tar.gz
```

Once you unzip this file, you can start phometa server by execute

```
./phometa-server.py 8080
```

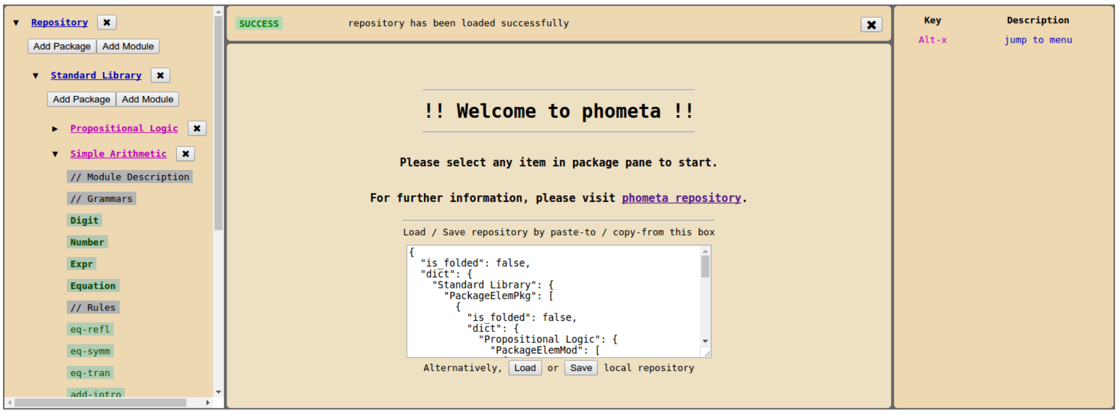
where `8080` is port number, you can change this to another port number if you like. Please note that python is required for this server.

Then open your favourite web-browser[1] and enter

```
http://localhost:8080/phometa.html
```

The program will look like this

[1] but Google Chrome is recommended



SUCCESS repository has been loaded successfully

!! Welcome to phometa !!

Please select any item in package pane to start.

For further information, please visit phometa repository.

Load / Save repository by paste-to / copy-from this box

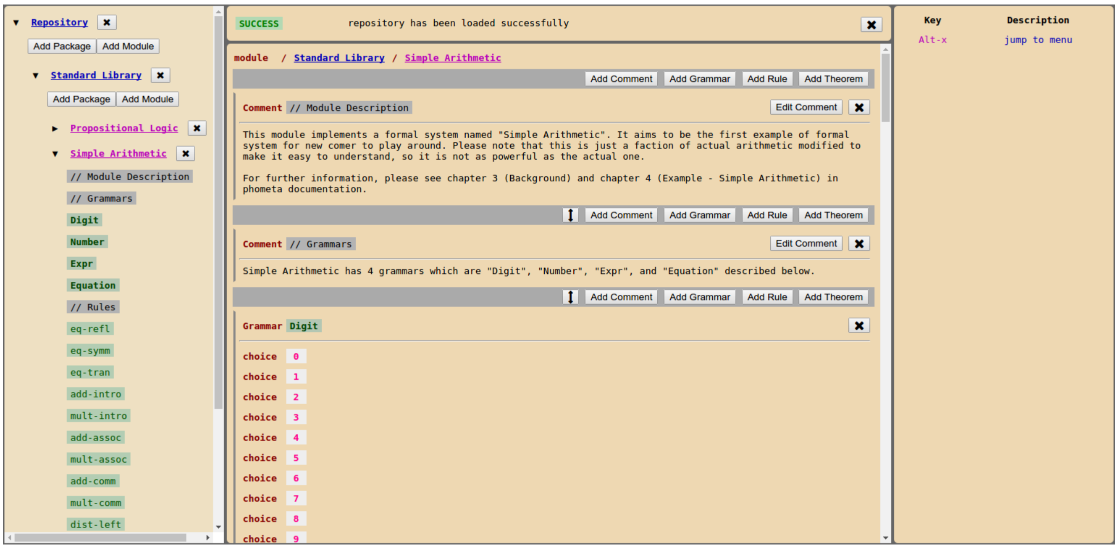
```
{
  "is_folded": false,
  "dict": {
    "Standard Library": {
      "PackageElemPkg": [
        {
          "is_folded": false,
          "dict": {
            "Propositional Logic": {
              "PackageElemMod": [
```

Alternatively, Load or Save local repository

Figure 4.1: Screenshot of Phometa when you open it from web-browser.

Phometa has a repository which consists of packages and modules that store formal systems and its proofs. The left pane of figure 4.1 shows global structure of a repository. The current repository has one package named “Standard Library” which consists of three modules named “Propositional Logic”, “Simple Arithmetic”, and “Typed Lambda Calculus”.

Module in phometa are analogous to text file. It consists of nodes that could depend on one another. There are four types of node which are *Comment*, *Grammar* (Backus-Naur Form), *Rule* (Derivation Rule), and *Theorem* (Derivation Tree). If you click at a module on the repository pane e.g. “Simple Arithmetic”, you will see the whole content of the module appear on the centre pane. Alternatively, you can click on each node on the repository pane directly to focus on particular node.



SUCCESS repository has been loaded successfully

Comment // Module Description

This module implements a formal system named "Simple Arithmetic". It aims to be the first example of formal system for new comer to play around. Please note that this is just a faction of actual arithmetic modified to make it easy to understand, so it is not as powerful as the actual one.

For further information, please see chapter 3 (Background) and chapter 4 (Example - Simple Arithmetic) in phometa documentation.

Comment // Grammars

Simple Arithmetic has 4 grammars which are "Digit", "Number", "Expr", and "Equation" described below.

Grammar Digit

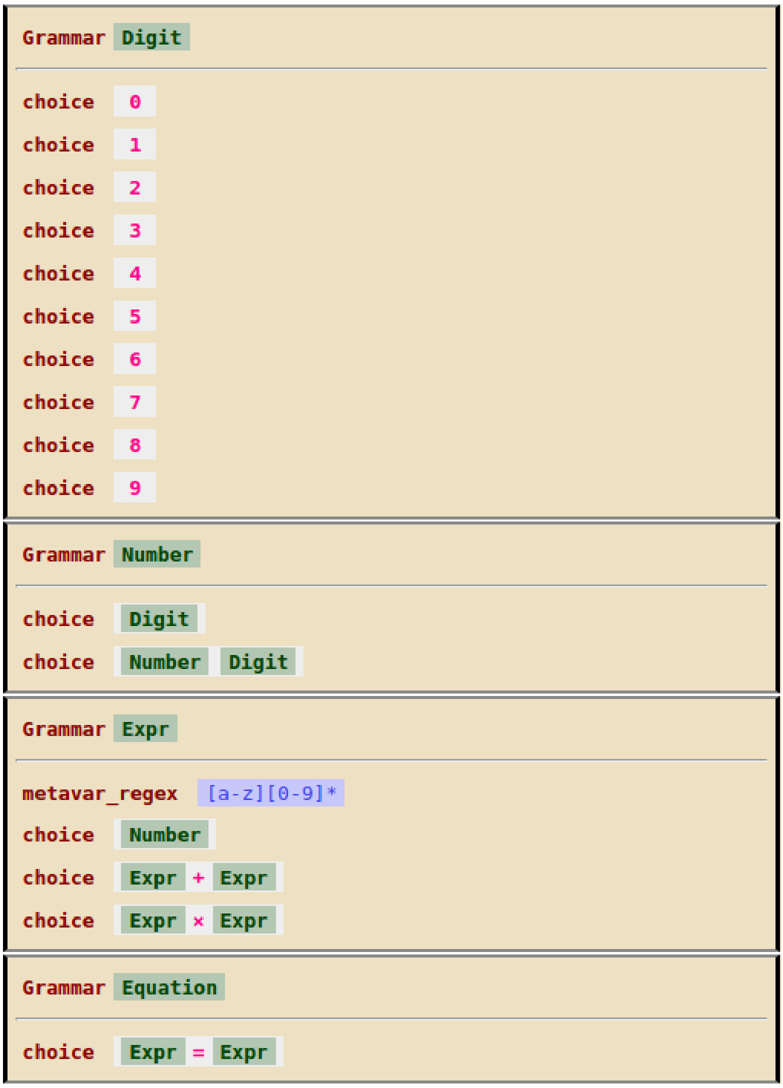
choice 0
choice 1
choice 2
choice 3
choice 4
choice 5
choice 6
choice 7
choice 8
choice 9

Figure 4.2: Screenshot of Phometa when you click “Simple Arithmetic” module.

In order to improve productivity, phometa has several key-bindings specific to curtain state of program. Fortunately, user don't need to remember any of this since the right pane (i.e. keymap pane) shows every possible key-binding with its description on current state. This also allow new-comer to explore new features during using it

4.2 Grammars

The Backus-Naur Form of Simple Arithmetic in figure 3.1 could be transformed in to this four following grammars



```
Grammar Digit

choice  0
choice  1
choice  2
choice  3
choice  4
choice  5
choice  6
choice  7
choice  8
choice  9

Grammar Number

choice  Digit
choice  Number Digit

Grammar Expr

metavar_regex  [a-z][0-9]*
choice  Number
choice  Expr + Expr
choice  Expr × Expr

Grammar Equation

choice  Expr = Expr
```

Figure 4.3: Grammars of Simple Arithmetic

We take advantage of visualisation by replacing brackets with underlines, this should improve readability because reader can see a whole term in a compact way but still able check how they are bounded when needed, for example,

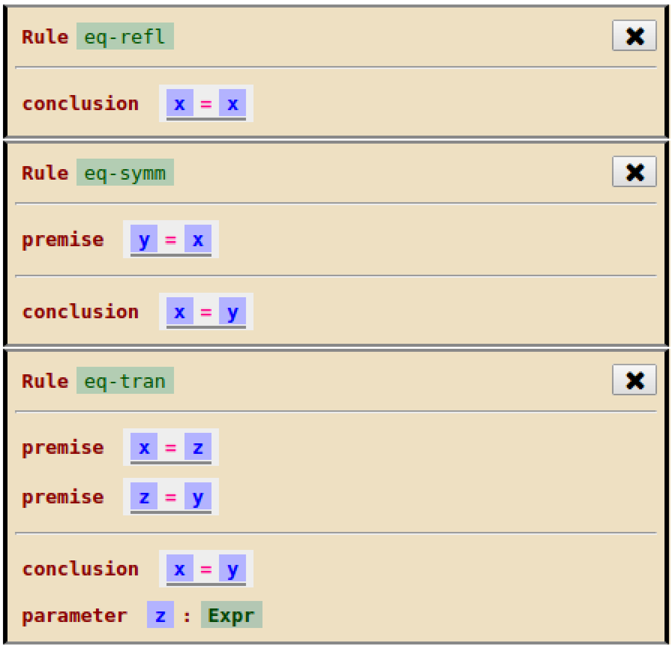
- `<Number>` 250 is transformed to `Number` `2 5 0`
- `<Expr>` $(0 + (12 \times 6))$ is transformed to `Expr` `1 2 + 0 × 6`
- `<Equation>` $(5 + 7) = 12$ is transformed to `Equation` `5 + 7 = 1 2`

These underline patterns coincide with diagram in figures 3.2, 3.3, 3.5, and 3.6 respectively.

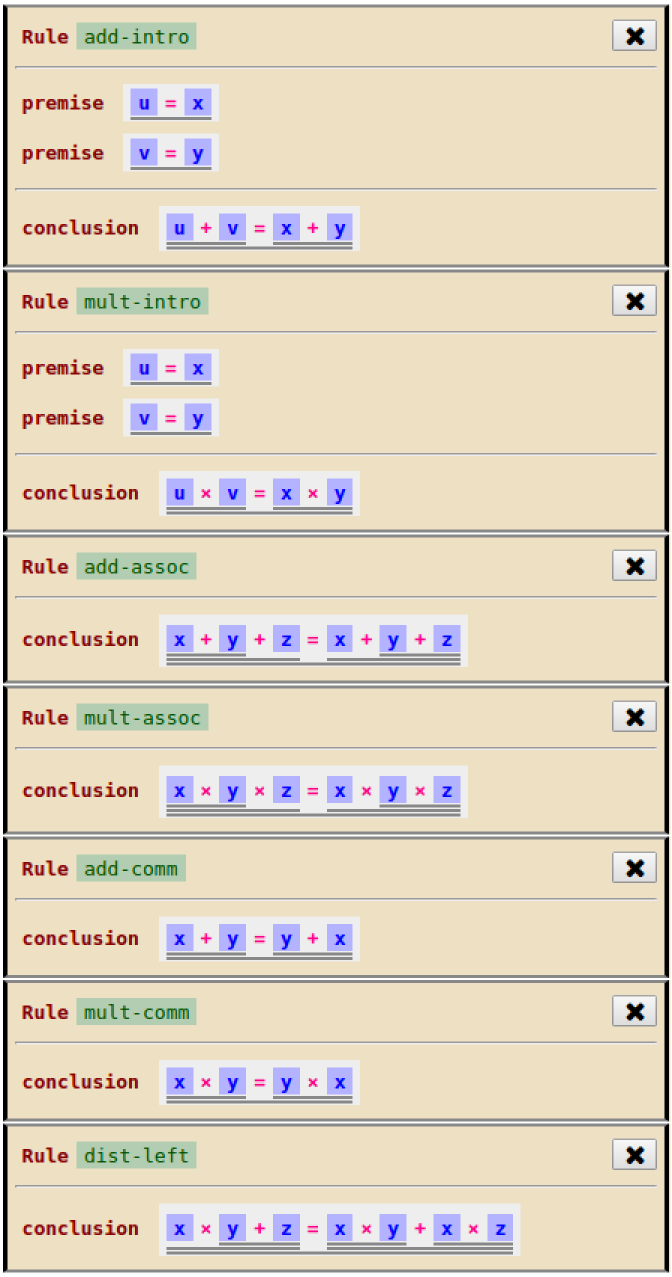
`metavar_regex` is used to control the name meta variables of each grammar. If this property is omitted, the corresponding grammar cannot instantiate meta variables. For example, `Expr` can instantiate meta variables with the name comply to regular expression `/[a-z][0-9]*/` (e.g. `a`, `b`, . . ., `z`, `a1`, `a2`, . . .), whereas `Digit`, `Number`, and `Equation` couldn't instantiate any meta variables, however, it could have meta variables as sub-term e.g. `Equation` `x + x = 2 × x`.

4.3 Rules

The derivation rules of Simple Arithmetic in figure 3.9 can be transformed as the following



```
Rule eq-refl                          ✖
conclusion  x = x

Rule eq-symm                          ✖
premise  y = x
conclusion  x = y

Rule eq-tran                          ✖
premise  x = z
premise  z = y
conclusion  x = y
parameter  z : Expr
```



```
Rule add-intro                              ✖
premise     u = x
premise     v = y
conclusion  u + v = x + y

Rule mult-intro                             ✖
premise     u = x
premise     v = y
conclusion  u × v = x × y

Rule add-assoc                              ✖
conclusion  x + y + z = x + y + z

Rule mult-assoc                             ✖
conclusion  x × y × z = x × y × z

Rule add-comm                               ✖
conclusion  x + y = y + x

Rule mult-comm                              ✖
conclusion  x × y = y × x

Rule dist-left                              ✖
conclusion  x × y + z = x × y + x × z
```

Figure 4.4: Rules of Simple Arithmetic

Most of rules here are self explain but in rule `eq-tran` , there is an additional property named `parameter` (s) which is a meta veritable that appear in premises but not in conclusion, hence user need to give a term when the rule is applied. Please note that `parameter` is automatic i.e. when user define they own rule, it will change automatically depending on premises and conclusion `dist-right` is not defined here but it will be defined as *lemma* in the next section.

4.4 Theorems and Lemmas

The first example of derivation tree (figure 3.11) could be transformed to theorem

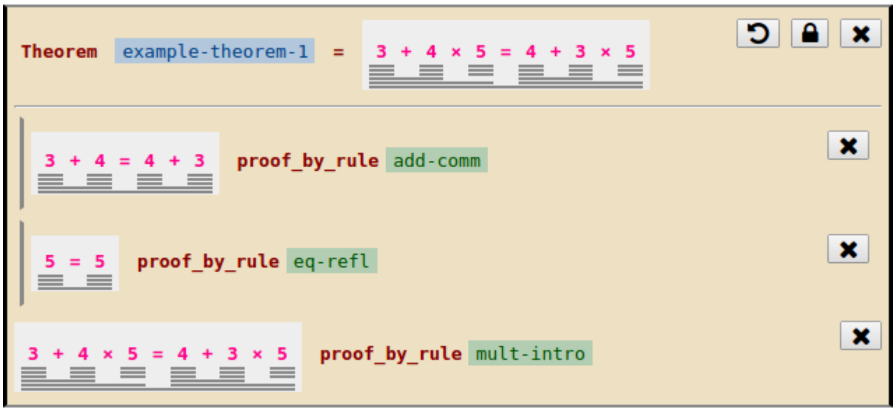
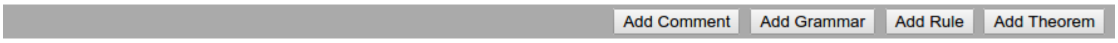


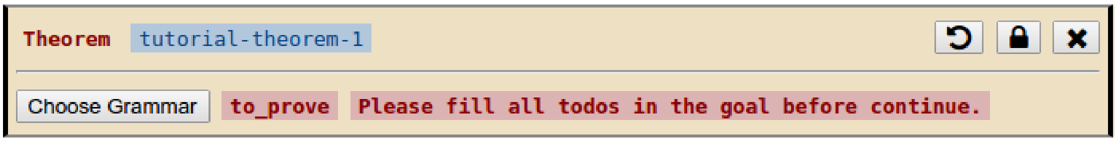
Figure 4.5: A theorem that show that $(3 + 4) \times 5 = (4 + 3) \times 5$.

You can see that the theorem still preserve tree-like structure but the width doesn't grow exponentially like derivation.

Next, I will show you that how was the theorem above constructed. Once we click module "Simple Arithmetic" on the repository pane, we will see the whole context similar to figure 4.2, you will also see that there are adding panel intersperse among each node

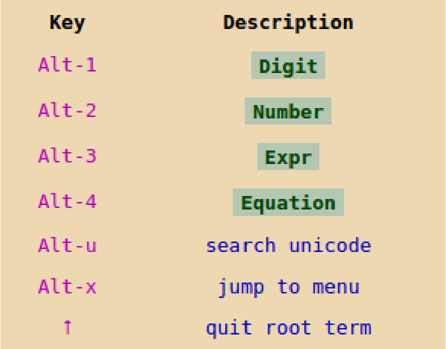


Now, click "Add Theorem", the button will change to input box where you can specify theorem name. Type "tutorial-theorem-1", can you will get empty theorem as the following



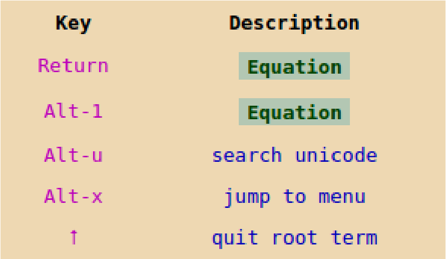
Please fill all todos in the goal before continue.

The first thing that we can do is to construct the goal that will be proven. On the picture above you will see button labelled "Choose Grammar" which is, in fact, a term that doesn't know its grammar. We can specify grammar by click the button, which in-tern, will change to input box. Now the keymap pane will look like this



Key	Description
Alt-1	Digit
Alt-2	Number
Alt-3	Expr
Alt-4	Equation
Alt-u	search unicode
Alt-x	jump to menu
↑	quit root term

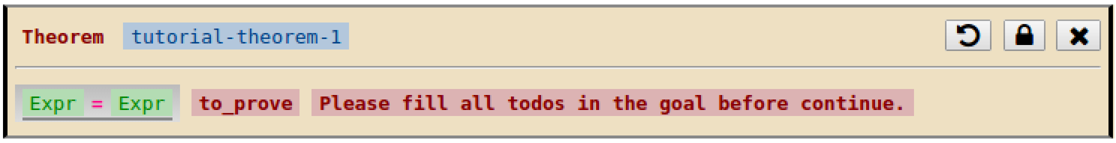
The keymap pane told us that there are 4 grammars available, we can either press Alt-1..4 or click on the row in keymap pane directly to select grammar. Alternatively, you can search a grammars by type faction on it is input box e.g. “eq”



Key	Description
Return	Equation
Alt-1	Equation
Alt-u	search unicode
Alt-x	jump to menu
↑	quit root term

Now, it is only Equation available because it is the only one that has “eq” as (case-insensitively) sub-string. And since it is the only one, you can select it by press Return , even though in this case we don’t have too many options but still benefit from it in auto complete favour. Please note that this input box support multiple matching separated by space e.g. “eq ti” still match Equation because both of “eq” and “ti” are sub-string of it.

Once you select a grammar, the box will change to green colour and waiting for a term of that grammar. In this case, we select Equation since it has just one choice and doesn’t have meta variable or literal[2] so phometa automatically click such a choice and the theorem will look like this



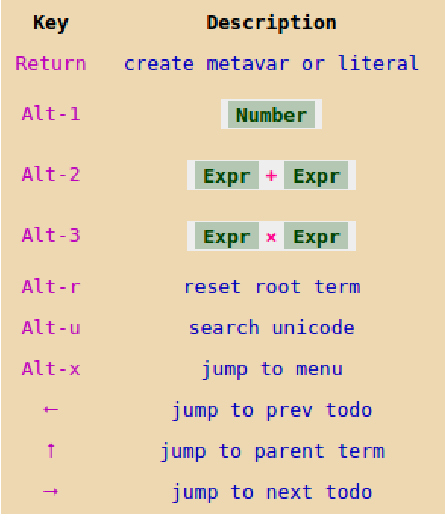
Theorem tutorial-theorem-1

Expr = Expr to_prove Please fill all todos in the goal before continue.

You may notice that the goal term as grey background rather than white as before. This indicate that the term is still modifiable.

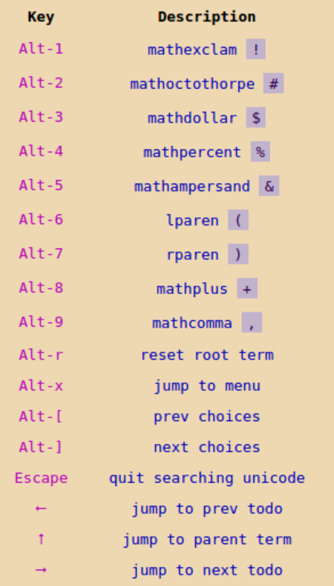
Next, we will continue on the Expr term on the left hand side of “=”. When the cursor is in it, the keymap will look like this

[2] literal is similar to meta-variable but only match to itself, will be explained in later chapter



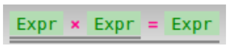
Key	Description
Return	create metavar or literal
Alt-1	Number
Alt-2	Expr + Expr
Alt-3	Expr × Expr
Alt-r	reset root term
Alt-u	search unicode
Alt-x	jump to menu
←	jump to prev todo
↑	jump to parent term
→	jump to next todo

Again, there are 3 choice available which can be selected similar manner when we select grammar. At the this stage you might wonder how to type "×" since it is unicode character. Well, we can go to unicode mode by press `Alt-u` as keymap pane suggest. Then keymap pane will look like this



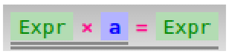
Key	Description
Alt-1	mathexclam !
Alt-2	mathoctothorpe #
Alt-3	mathdollar $
Alt-4	mathpercent %
Alt-5	mathampersand &
Alt-6	lparen (
Alt-7	rparen)
Alt-8	mathplus +
Alt-9	mathcomma ,
Alt-r	reset root term
Alt-x	jump to menu
Alt-[	prev choices
Alt-]	next choices
Escape	quit searching unicode
←	jump to prev todo
↑	jump to parent term
→	jump to next todo

This allow us to search unicode character by using its LaTeX's math-mode name. Now type "times" in the input box, you should see "×" appear on keymap. Once you select it, the unicode mode disappear and put "×" in the input box, which in-tern, filter other choices out so you can hit `Return` for multiplication. The goal will transform to

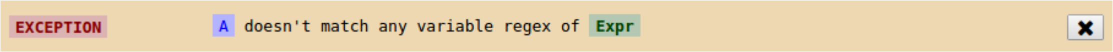


Expr × Expr = Expr

Next we will focus middle Expr . If we type string and hit Return here it will assume that we enter meta variable or literal (to avoid conflict auto complete similar to choose grammar is disabled here). E.g. if we type "a" and press enter it will become like this

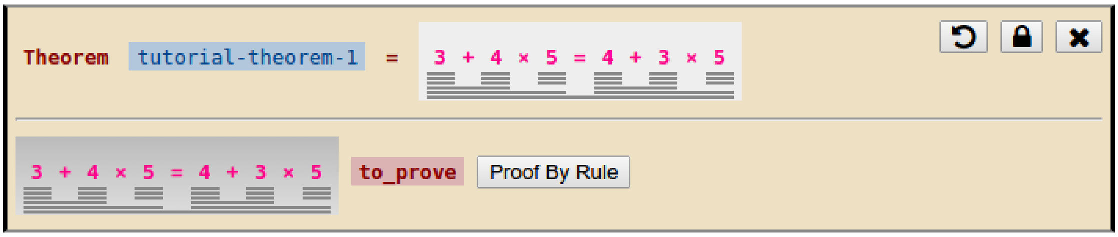

Expr × a = Expr

If we enter the name that that doesn't comply to regular expression, it will do nothing and prompt error message above main pane as the following


EXCEPTION A doesn't match any variable regex of Expr

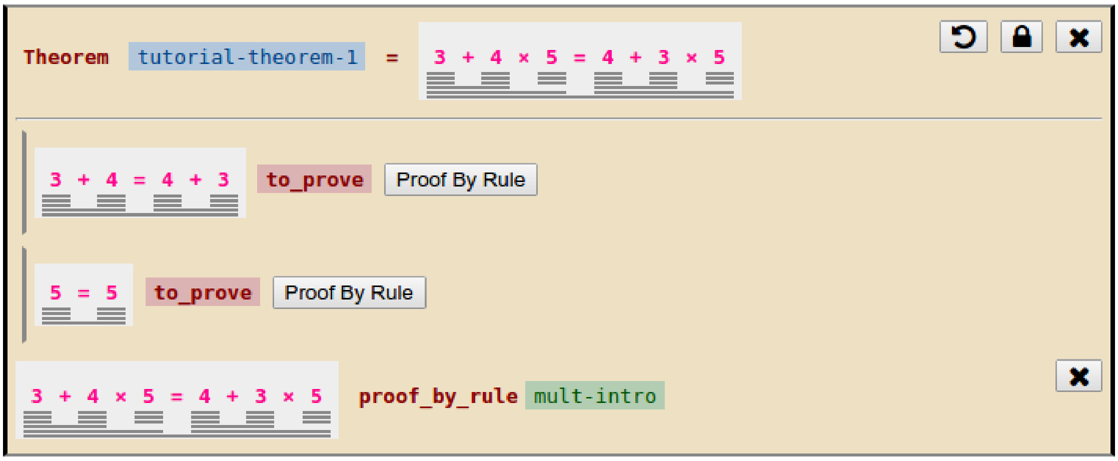
By the way, the goal here doesn't involve any meta variable. We can reset any sub-term (e.g. in this case a) by pressing Alt-t . Ultimately, you can reset the whole term by pressing Alt-r . In addition, you can jump to parent term by pressing UP , this is particularly useful when combine with Alt-t i.e. you can reset parent term only by keystrokes rather than clicking. You also be able to jump to previous todo or next todo by pressing LEFT or RIGHT respectively.

By recursively fill the the goal, eventually it will become like this


Theorem tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

3 + 4 × 5 = 4 + 3 × 5 to_prove Proof By Rule

Since the goal is complete, it is ready to be proven. You can select a rule by clicking "Proof By Rule" and select mult-intro similar manner to choose grammar. Then the theorem will look like this


Theorem tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

3 + 4 = 4 + 3 to_prove Proof By Rule

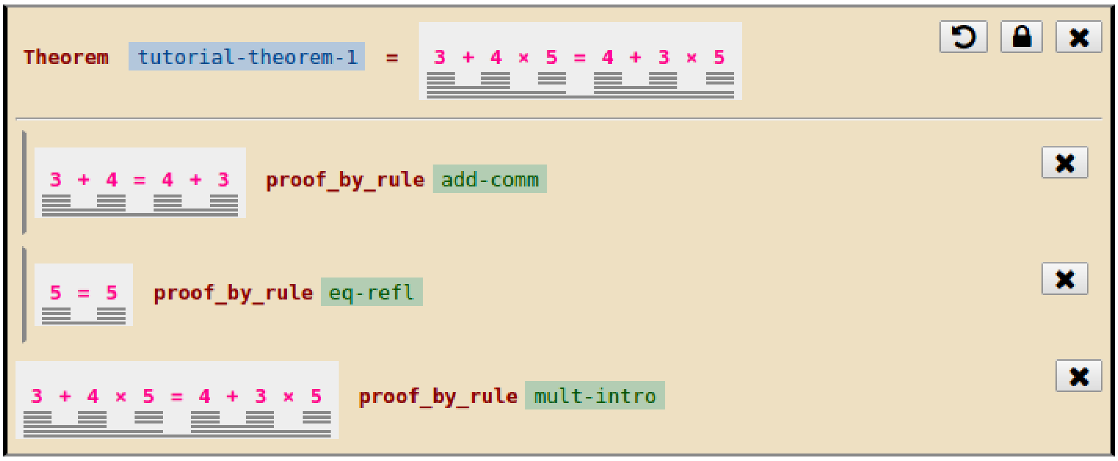
5 = 5 to_prove Proof By Rule

3 + 4 × 5 = 4 + 3 × 5 proof_by_rule mult-intro

Once the rule is applied, it will generate further sub-goals.

Please notice that the goal background changes to white colour as we can longer modify the goal. However if you made a mistake and want to go back, you can click close button on the bottom right corner of current proof (or press `Alt-t`) to reset current proof then you can modify it again. Ultimately, you can click reset button on the top right corner of the theorem (or press `Alt-r`) to reset entire theorem.

Two remaining sub-goals that have been generated can be proven similar to the process above i.e. select rule `add-comm` for first sub goal and `eq-refl` for second one.



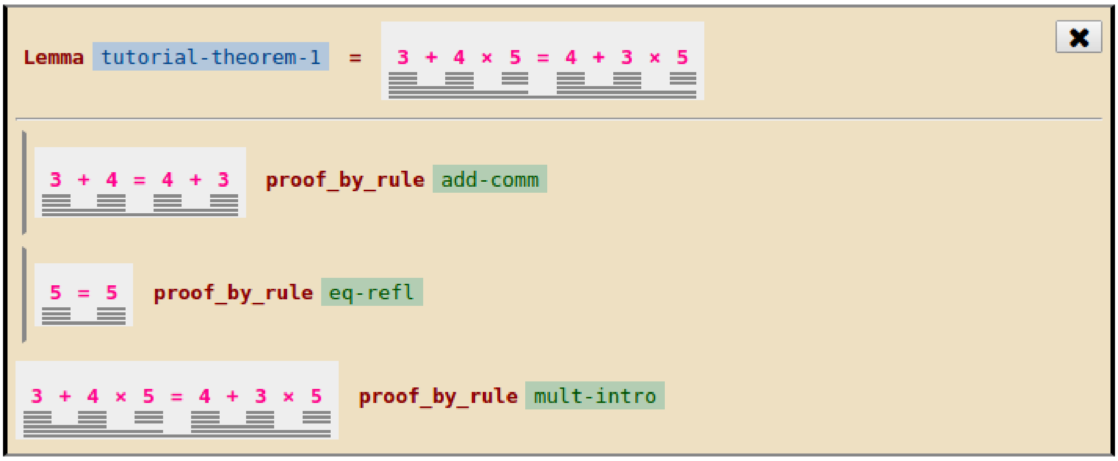
Theorem tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

3 + 4 = 4 + 3 proof_by_rule add-comm

5 = 5 proof_by_rule eq-refl

3 + 4 × 5 = 4 + 3 × 5 proof_by_rule mult-intro

Once the theorem is complete, you can claim validity of the goal. More over you can convert it to lemma that can be used in later theorem by clicking lock button on top-right corner of theorem



Lemma tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

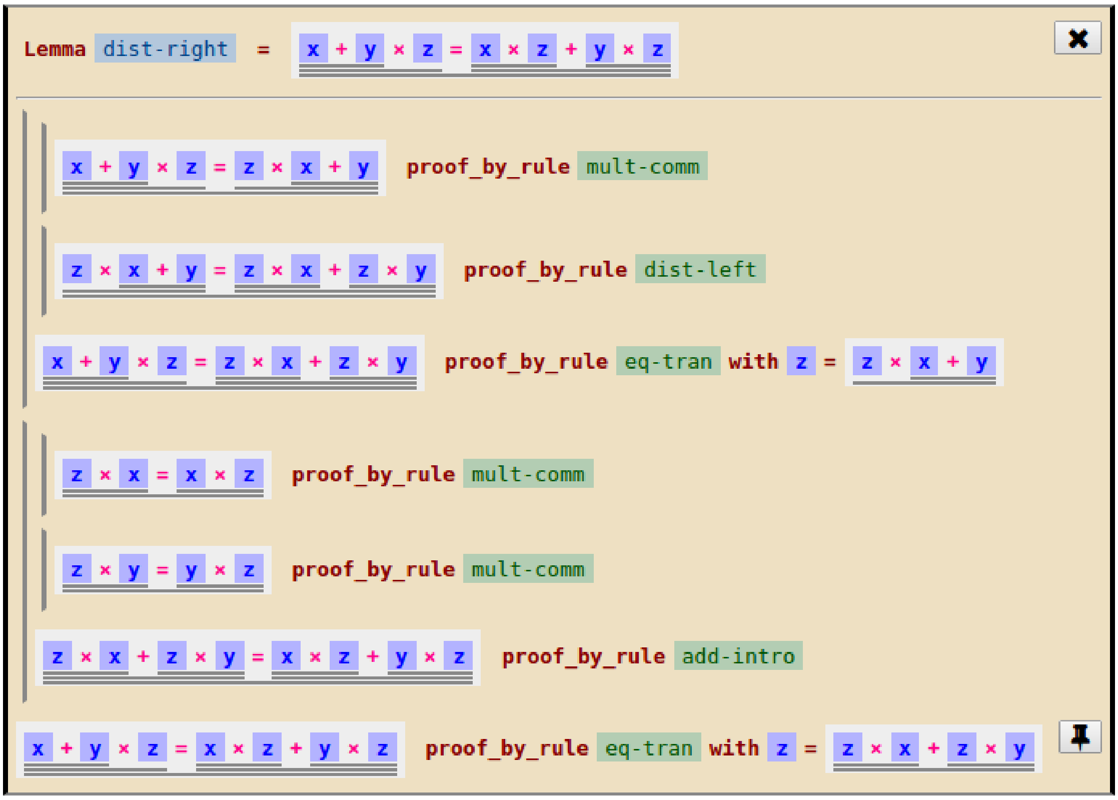
3 + 4 = 4 + 3 proof_by_rule add-comm

5 = 5 proof_by_rule eq-refl

3 + 4 × 5 = 4 + 3 × 5 proof_by_rule mult-intro

Because other theorem can use this lemma so it is no longer modifiable, as you can see that close button of each sub-proof and reset button of the main theorem are gone.

Similarly, we can can create lemma dist-right as the following



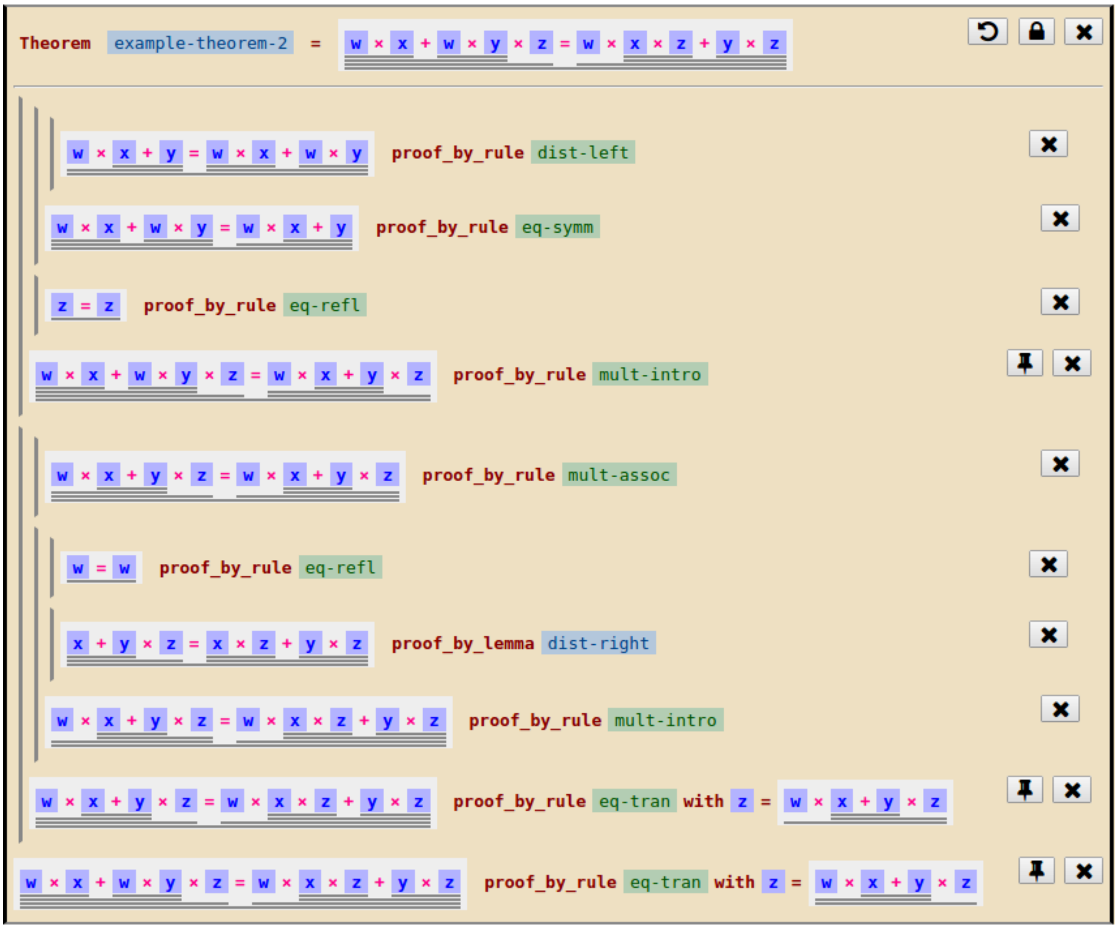
```
Lemma dist-right = x + y × z = x × z + y × z

x + y × z = z × x + y  proof_by_rule mult-comm
z × x + y = z × x + z × y  proof_by_rule dist-left
x + y × z = z × x + z × y  proof_by_rule eq-tran with z = z × x + y
z × x = x × z  proof_by_rule mult-comm
z × y = y × z  proof_by_rule mult-comm
z × x + z × y = x × z + y × z  proof_by_rule add-intro
x + y × z = x × z + y × z  proof_by_rule eq-tran with z = z × x + z × y
```

Figure 4.6: An example of lemma obtained by lock a theorem.

It is a good practice to create lots of small lemmas rather than a big theorem. This is because it is easier to read and you can use a lemma multiple time i.e. no need to duplicate sub-proof.

4.5 More complex theorem

To gain more familiarly on theorem, here is more complex theorem corresponded the second example of derivation tree on figure 3.12



Theorem example-theorem-2 = w × x + w × y × z = w × x × z + y × z

w × x + y = w × x + w × y proof_by_rule dist-left

w × x + w × y = w × x + y proof_by_rule eq-symm

z = z proof_by_rule eq-refl

w × x + w × y × z = w × x + y × z proof_by_rule mult-intro

w × x + y × z = w × x + y × z proof_by_rule mult-assoc

w = w proof_by_rule eq-refl

x + y × z = x × z + y × z proof_by_lemma dist-right

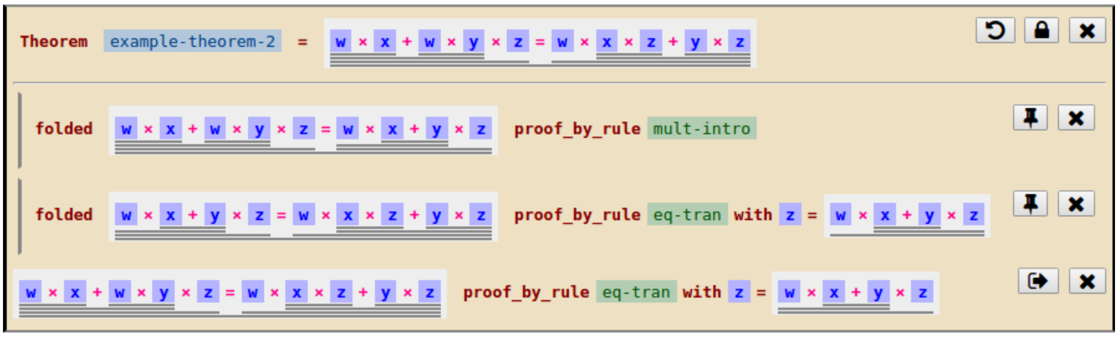
w × x + y × z = w × x × z + y × z proof_by_rule mult-intro

w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

Figure 4.7: The second example theorem of Simple Arithmetic.

Sub-proofs become more complex and premises of main proof are far away which is harder to read, to avoid this kind of problem, we introduce a focus button that will fold sub-proof of corresponding proof. For example, if we click focus button on the main proof it will look like this



Theorem example-theorem-2 = w × x + w × y × z = w × x × z + y × z

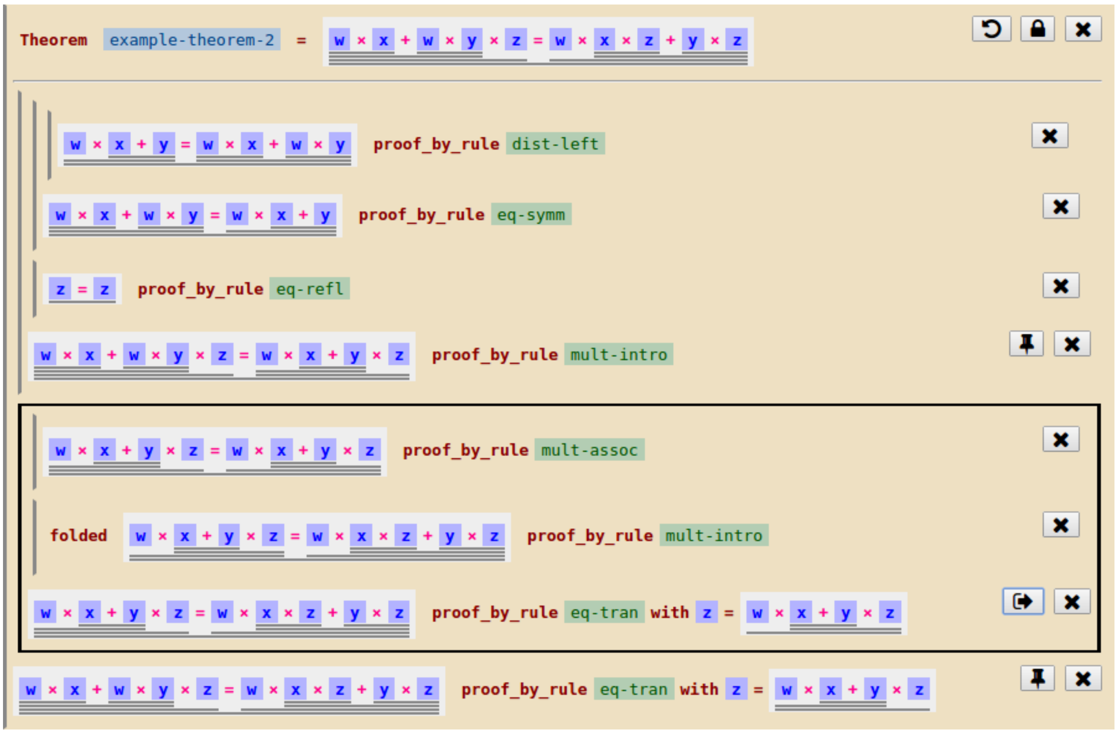
folded w × x + w × y × z = w × x + y × z proof_by_rule mult-intro

folded w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

Two sub-proofs of the main theorem are folded, this allow us to read rule instance of eq-tran easily. You can unfolded sub-proofs by clicking unfocus button at the same position that focus button was there before. And the theorem will look the same as figure 4.7 again.

When you read some proof in phometa, it is a good idea to click focus on the main proof first so you can read the main rule instance easily. Once you understand main proof you can read one of sub proof by clicking focus button that correspond to that sub-proof. If you click focus button on the first sub-proof it will look like this



Theorem example-theorem-2 = w × x + w × y × z = w × x × z + y × z

w × x + y = w × x + w × y proof_by_rule dist-left

w × x + w × y = w × x + y proof_by_rule eq-symm

z = z proof_by_rule eq-refl

w × x + w × y × z = w × x + y × z proof_by_rule mult-intro

w × x + y × z = w × x + y × z proof_by_rule mult-assoc

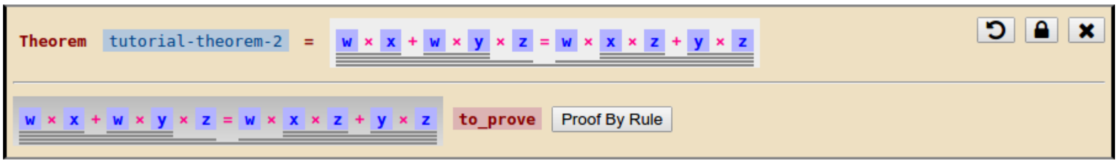
folded w × x + y × z = w × x × z + y × z proof_by_rule mult-intro

w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

This process automatically unfold the previous one before folding sub-proof of this proof again. Please note that some of deeper proofs might not have focus button at all, this is because its sub-proofs cannot reduce further than original one.

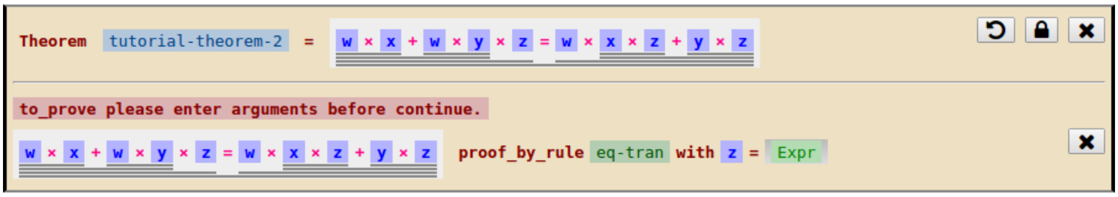
The next thing that I will show are how to use rule parameters and lemma. This can be illustrated by recreate this theorem again. First create create a theorem tutorial-theorem-2 using the same goal as above theorem.



Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

w × x + w × y × z = w × x × z + y × z to_prove Proof By Rule

Then apply rule eq-tran to this goal

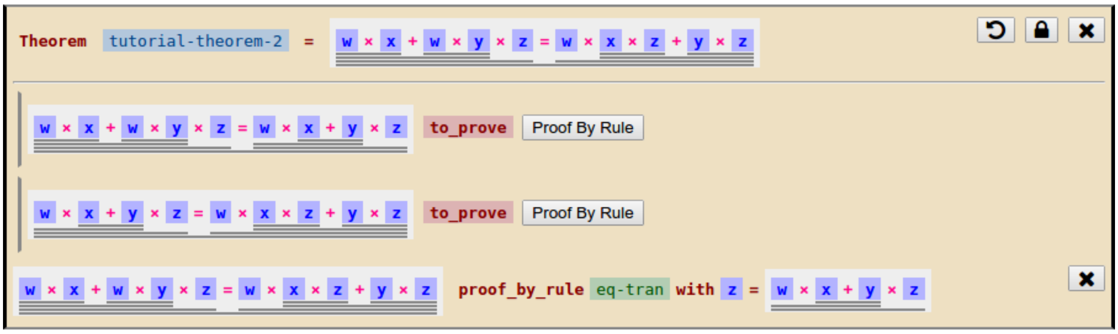


Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

to_prove please enter arguments before continue.

w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = Expr

The rule applying process is not complete because eq-tran contains z which appear in premises but not in conclusion (i.e. z is parameter) so phometa ask us to fill the term that we want to use. In this case we want w × x + y × z so put it there



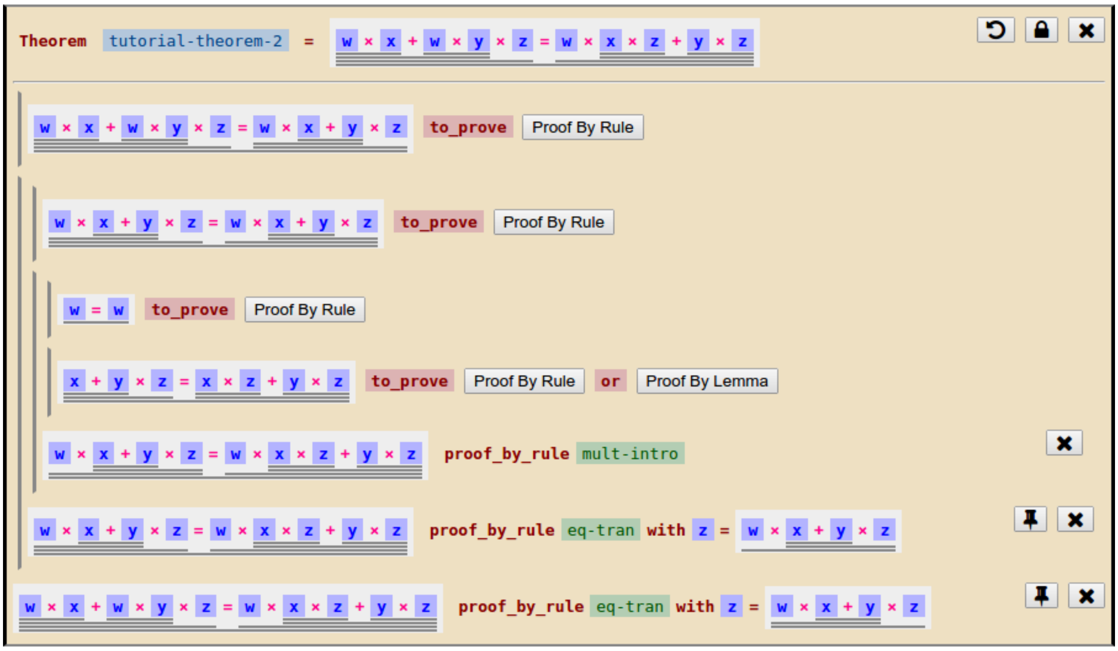
Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

w × x + w × y × z = w × x + y × z to_prove Proof By Rule

w × x + y × z = w × x × z + y × z to_prove Proof By Rule

w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

Now, let focus on the second sub-goal, we can apply eq-tran again but with z = w × x + y × z. Then, apply mult-intro in the second its sub-goal.



Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

w × x + w × y × z = w × x + y × z to_prove Proof By Rule

w × x + y × z = w × x + y × z to_prove Proof By Rule

w = w to_prove Proof By Rule

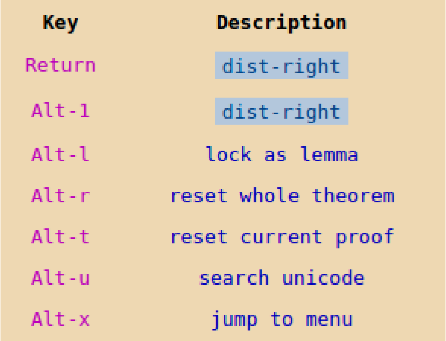
x + y × z = x × z + y × z to_prove Proof By Rule or Proof By Lemma

w × x + y × z = w × x × z + y × z proof_by_rule mult-intro

w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

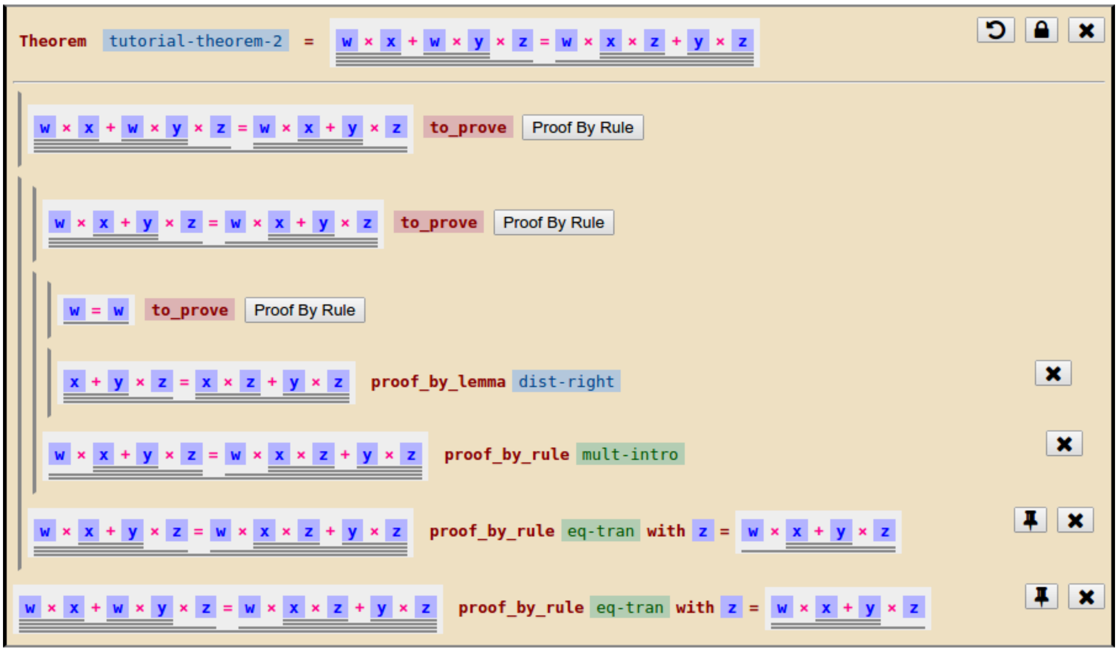
w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

You can see that there is a sub-goal that has button “Proof By Lemma”. This is because there is at least one lemma that is pattern matchable with that sub-goal. If you click “Proof By Lemma” button, the keymap will look like this



Key	Description
Return	dist-right
Alt-1	dist-right
Alt-l	lock as lemma
Alt-r	reset whole theorem
Alt-t	reset current proof
Alt-u	search unicode
Alt-x	jump to menu

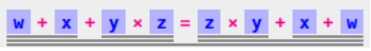
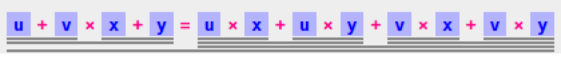
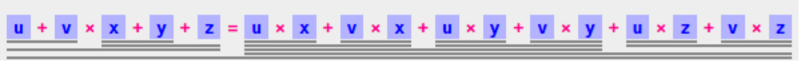
In this case, there is one lemma which is `dist-right` that is pattern matchable to that sub-goal. You can hit `Return` to use this lemma and it will look like this



```
Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z
w × x + w × y × z = w × x + y × z  to_prove  Proof By Rule
w × x + y × z = w × x + y × z  to_prove  Proof By Rule
w = w  to_prove  Proof By Rule
x + y × z = x × z + y × z  proof_by_lemma dist-right
w × x + y × z = w × x × z + y × z  proof_by_rule mult-intro
w × x + y × z = w × x × z + y × z  proof_by_rule eq-tran with z = w × x + y × z
w × x + w × y × z = w × x × z + y × z  proof_by_rule eq-tran with z = w × x + y × z
```

The remaining step is easy enough.

4.6 Exercises

- Create a theorem and proof each of the following
  - 
    w + x + y × z = z × y + x + w
  - 
    u + v × x + y = u × x + u × y + v × x + v × y
  - 
    u + v × x + y + z = u × x + v × x + u × y + v × y + u × z + v × z
- Extend Simple Arithmetic to support
  - addition and multiplication identity.
  - addition and multiplication idempotent.
  - inequality.
- Create a theorem of your own choice and proof it.

Chapter 5

Example Formal System — Propositional Logic

Once you are familiar with some basic features and usability of *phometa* from the last chapter. This chapter aims to show more advance features on another formal system named *Propositional Logic* which is the most well known logical system[1].

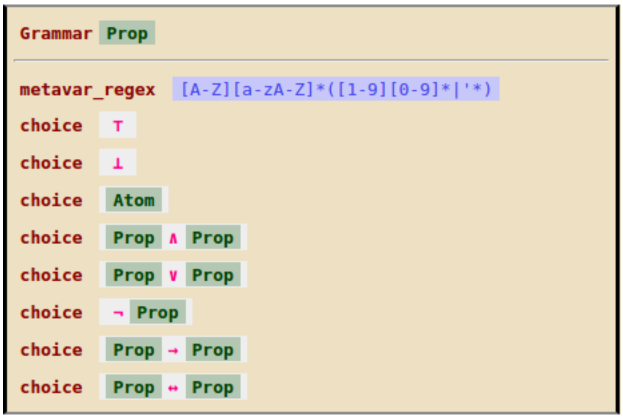
Logic, in general, works so well with traditional derivation system, hence there is spacial name called *Natural Deduction* which combination of any kind of Logic together with derivation system.

5.1 Grammars

As usual, the first thing that needed to be defined is syntax which consists of several grammars. Propositional logic has `Prop` , `Atom` , `Context` , and `Judgement` as its grammars.

`Prop` is a proposition, semantically, it is a term that can be evaluated to either true or false (given that there are no variables in the term). Grammars of `Prop` can be defined in phometa as the following

[1] Logical system is a formal system together with semantics[14]



```
Grammar Prop

metavar_regex  [A-Z][a-zA-Z]*([1-9][0-9]*|'*)
choice  ⊤
choice  ⊥
choice  Atom
choice  Prop ∧ Prop
choice  Prop ∨ Prop
choice  ¬ Prop
choice  Prop → Prop
choice  Prop ↔ Prop
```

This grammar is equivalence to the following Backus Normal Form

```
<Prop> ::= ⊤ | ⊥ | <Atom>
         | <Prop> ∧ <Prop>
         | <Prop> ∨ <Prop>
         | ¬ <Prop>
         | <Prop> → <Prop>
         | <Prop> ↔ <Prop>
         | meta-variables comply with regex
             /[A-Z][a-zA-Z]*([1-9][0-9]*|'*)/
```

We take advantage of visualisation by replacing brackets with underlines, this should improve readability because reader can see a whole term in a compact way but still able check how they are bounded when needed, for example,

- $\underline{\top}$ represents $(\top)$
- $\underline{\bot}$ represents $(\bot)$
- $\underline{\top}\ \underline{\wedge}\ \underline{\bot}$ represents $((\top) \wedge (\bot))$
- $\underline{\bot}\ \vee\ \underline{\underline{\top}}\ \underline{\vee}\ \underline{\bot}$ represents $((\bot) \vee ((\top) \vee (\bot)))$
- $\underline{\top}\ \underline{\wedge}\ \underline{\underline{\top}}\ \rightarrow\ \underline{\bot}$ represents $(((\top) \wedge (\top)) \rightarrow (\bot))$
- and so on ...

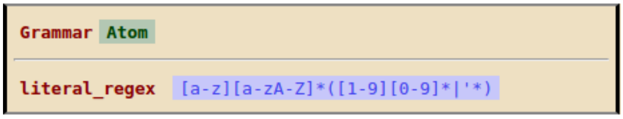
Having only close terms is not so useful, so we allow terms to have meta-variables, when this is the case, we can see these terms as *pattern*, for example,

- A represents arbitrary Prop term
- $\underline{\neg}$ A represents Prop term that has ¬ as main connector
- A $\underline{\wedge}$ B represents Prop term that has ∧ as main connector

- A ∧ A represents Prop term that has ∧ as main connector and both of its sub-terms are identical
- and so on ...

Meta-variables names must comply to regular expression stated in var_regex of corresponding grammar declaration, this allow us to enforce some naming convention, for example, meta-variables of Prop must start with capital letter.

Next, we would like to create the grammar for Atom . Intuitively, an atom is primitive truth statement that can be either true or false. we can define Atom like the following.



```
Grammar Atom
literal_regex [a-z][a-zA-Z]*([1-9][0-9]*|'*)
```

Atom definition doesn't neither metavar_regex or choice s

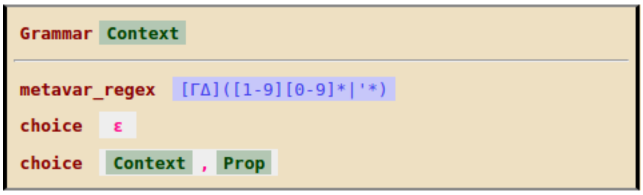
we can represent it as a grammar that doesn't have any choice, hence, using meta-variable become the only way to instantiate Atom . (TODO: replace this with the real screenshot)

At this stage, reader might wonder that what is different between meta-variable of Prop and Atom choice in Prop , well, meta-variable is something that can be substituted later while Atom it truth statement that is static. For example, if we know that A ∨ ¬ A holds, we can derive corresponding term such as ⊤ ∨ ¬ ⊤ , raining ∨ ¬ raining , B ∧ C ∨ ¬ B ∧ C directly, but knowing raining ∨ ¬ raining doesn't help us to derive something like the former case does. In order to prevent this confusion we set var_regex of Prop and Atom differently, so if user accidentally write p on a hole that require Prop , phometa will raise an error messages and user can fix it.

(TODO: ask Dr Krysia, the reason above doesn't prevent meta-variable of atom to be substituted by another meta-variable of atom, this problem also rise when I try to combine hypothesis-base and hypothesis next together. I am thinking to create another kind of meta-variable that doesn't allow substitution (maybe under name name_regex) such that meta-variable under this regular expression cannot be substituted)

Now, we have enough ingredient to create a proper proposition, one might say that we can start proving it directly, however, most of proposition that we will dealing with only holds under certain assumptions, hence, a *judgement* should be in the form $A_1, A_2, ..., A_n \vdash B$ where $A_{1..n}$ are assumptions and B is conclusion.

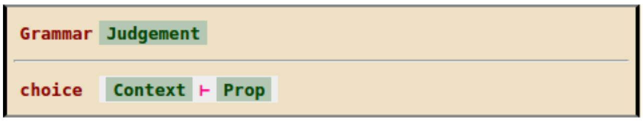
To model a judgement in phometa, first we need to model assumptions or in the other name, Context as the following (TODO: replace this with the real screenshot)



```
Grammar Context

metavar_regex  [ΓΔ]([1-9][0-9]*|'*)
choice  ε
choice  Context , Prop
```

So a term of Context can be either empty context or another context appended by a proposition. We can see a context as a list of proposition.

Now we are ready to define Judgement as the following (TODO: replace this with the real screenshot)



```
Grammar Judgement

choice  Context ⊢ Prop
```

Examples of Judgement term could be like

- ϵ , p , p → q ⊢ q

  assume atom p and proposition p → q then atom q holds

- ϵ , B ∧ A ⊢ A ∧ B

  let A and B be arbitrary Prop and assume that B ∧ A then A ∧ B holds

- ϵ ⊢ A ∨ ¬ A

  let A be arbitrary Prop then A ∨ ¬ A holds

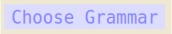
Please note that Judgement doesn't have field var_regex so we can't accidentally use meta-variable for Judgement .

5.2 Input method of a term

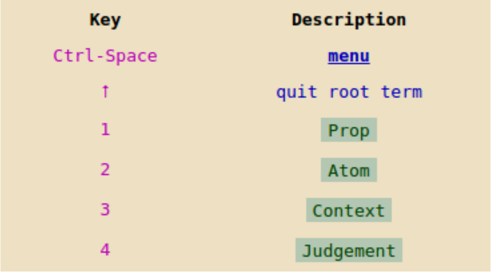
When a term is created, it can be in one of this mode

5.2.1 Editable up to Grammars

On this mode, we have a absolute control over a term i.e. both of grammar and term content can be changed, when it is created it will look like this,

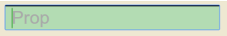


This is a completely blank term, we can set its grammar by clicking on it, then you will see option on keymap-pane (left-bottom corner of screen) as the following



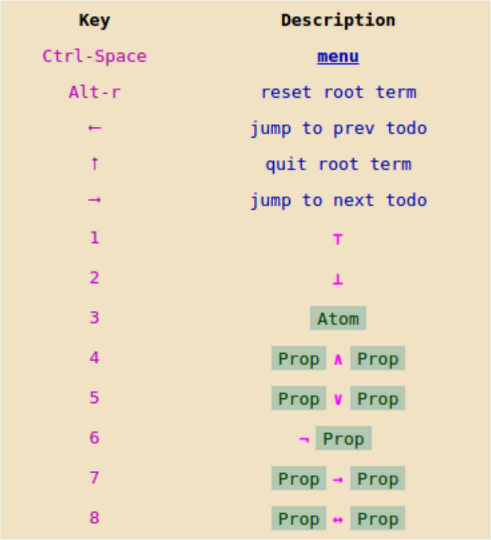
Key	Description
Ctrl-Space	menu
↑	quit root term
1	Prop
2	Atom
3	Context
4	Judgement

This show all of possible key-blinding including all of grammar that can be use, now we will select Prop for this term's grammar by press *1* (or alternatively clicking on Prop directly), which now tern the term to be like this



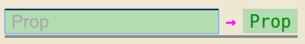
Prop

This means the term now know that it is Prop term and hasn't know a content, and in order give a content we can set it to be either a construction term or meta-variable term. Now let make it a construction term, if you see keymap-pane again, it will look like this



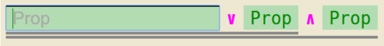
Key	Description
Ctrl-Space	menu
Alt-r	reset root term
←	jump to prev todo
↑	quit root term
→	jump to next todo
1	⊤
2	⊥
3	Atom
4	Prop ∧ Prop
5	Prop ∨ Prop
6	¬ Prop
7	Prop → Prop
8	Prop ↔ Prop

Similar to grammar selection, we can press the corresponded key-blinding that construct the term as we like. Let press *7* for Prop → Prop and now the term become like this,



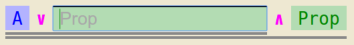
Prop → Prop

The root term now has → as the main connector and *todos* spilt into two place, this process is recursive, you can try to split it again by e.g. press *5* for Prop ∧ Prop and get

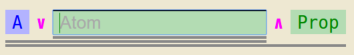


We also be able to move to other todos by clicking on target todo or simply press left arrow or right arrow for previous and next todo respectively.

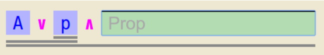
Next, we can make it to be meta-variable term by enter meta-variable name on the target todo then press enter. Let type *A* for the first todo



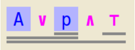
Now the first term become meta-variable and the cursor move to the next one automatically, please note that the meta-variable name must comply to `var_regex` of corresponding grammar, otherwise, it will not convert to meta-variable. If you want to create an atom here, you can't just type directly since it is for `Prop` meta-variable (As you can see from background text in todo), instead, you need to press *3* to tell phometa that it is construction for `Atom` , like this



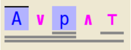
Now the background of current todo change from *Prop* to *Atom* and has extra underline so now you can type and atom



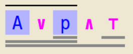
Not only meta-variable that can finish up todos, some of grammar choice just doesn't have sub-term, we could finish the last todo, by just press *1* for ⊤



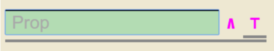
Now the term is finished, you can still return to the term and click some of them e.g. clicking A



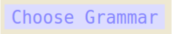
You can navigate to parent term my press up-arrow



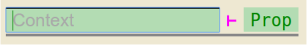
If you think you create some sub-term wrongly, you click on the that sub-term (in this is case A ∨ p) then press *Alt-t*



Or if everything is wrong including the root term grammar, you just press *Alt-r*, this will go back to *asking for grammar state*



To give more example, let try to create `Judgement` term, by click *4*



Context ⊢ Prop

You might expect that after pressing *4*, it should be just green area with background written as *Judgement*, why it is auto expend like this, well, we have spacial case that if selecting grammar as just one choice and can't be meta-variable (we didn't state `var_regex` for `Judgement`), it will auto expend to that choice since that is the only action available.

5.2.2 Editable up to Term

This mode is the the same as editable up to grammar except that the grammar is given and we can't change it. If we try to reset the whole term *Alt-r*, it the grammar still remain there.

5.2.3 Read only

In this mode, you can just view the term and nothing else.

5.3 The first theorem

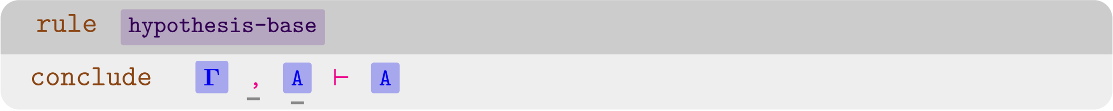
Grammars provide rigorous way to construct well-formed terms, however these terms doesn't have any meaning, the way to give the its meaning to prove that it is *valid*. Validity of each term depends on its grammars, for example, a `Prop` term is valid iff that term always evaluate to true, a `Judgement` term is valid iff assuming all propositions form left hand side then the proposition on the right hand side hold.

The main idea of proving in phometa is that you state (in a theorem) that some term is valid, then give a reason why it is valid, this can be done by either *Prove by Rule* or *Prove by Lemma*.

A rule is a meta-statement which say that, if some terms (premises) is valid then the conclusion term is valid.

We want ability to state that for any proposition that is in assumptions, it can be conclusion i.e. $\boldsymbol{A_{1..n} \vdash A_i}$ where $\boldsymbol{i \in 1..n}$

This is achievable by `hypothesis-base` and `hypothesis-next`



```
rule  hypothesis-base
conclude  Γ , A ⊢ A
```

This state that if we have a `Judgement` such that the last proposition in context is identical to the conclusion proposition, then that `Judgement` is valid.

```
rule      hypothesis-next
premise   Γ ⊢ A
conclude  Γ , B ⊢ A
```

This state that if we have a valid `Judgement` , then appending the another proposition in the context still valid

So let prove the very first judgement `ϵ , A ⊢ A`

TODO: continue here

At this stage, you might think that validity is necessary only for the grammars that have a truth value, e.g. it doesn't make sense to prove a `Context` term. Well, this is because we know that every well-form `Context` term is valid term. Generally, this is not always be the case, for example, if we define an algebra of integer like this

```
grammar    Int
var_regex  /a-z/
choice     0
choice     1
choice     Int + Int
choice     Int - Int
choice     Int × Int
choice     Int ÷ Int
```

Obviously, `Int` doesn't have truth value, nevertheless, not every `Int` term is valid e.g. `n ÷ 0` , so we need to provide rules to *verify* `Int` .

Chapter 6

Example Formal System - Lambda Calculus

Since the last two chapters show most of features and usability of *phometa* already so this chapter aims to show that phometa is powerful enough as it can even encode more complex formal system like *typed lambda calculus.* Hence, it is clear that phometa is suitable to encode most of formal system that user can think of.

6.1 Untyped Lambda Calculus

6.2 Simply-typed Lambda Calculus

Chapter 7

Specification

In this chapter, we will the full detail of phometa.

7.1 Overview

mainly talk about phometa UI structure, grids, and keymap pane

7.2 Repository

structure of repository

- add new sub-package or module inside a package (using package pane)
- load/save repository using textarea in home pane + also talk about stdlib
- add/swap nodes inside module
- dedicate view for each node (using package pane)

7.3 Node Comment

7.4 Node Grammar

7.5 Root Term

7.6 Node Rule

7.7 Node Theorem

Chapter 8

Implementation

This part we will talk about the implementation of phometa which is written in elm hosted at `https://github.com/gunpinyo/phometa`.

8.1 Decision on programming language

Elm[1] is a functional reactive programming language. It allows programmer to create web application by declaratively coding in Haskell-like language then compile the program to JavaScript. For more information, please see elm official website at `elm-lang.org`.

One of the most attractive feture of elm is its reactivity. This idea introduces a new data type called "signal" which is a data type that can change over time. For example, let c be a signal of integer defined as $a + b$ where a and b are other signal of integers. If $a = 2$ and $b = 3$, then surely $c = 5$. If later a is change to 4, then c will got automatically updated to 7.

Reactivity work very well with functional paradigm since all variables are immutable, so it is impossible for the program to be in inconsistent state in the sense that programmer forget to update value. In fact, this can lead to a good fit of model-controller-view (MCV) architecture.

Here are summery of reasons why I choose elm to implement phometa,

- Phometa is a web application, and elm is created to build something like this
- Phometa mainly dealing with declarative object, it is better to use funcation language to build it.
- Elm by its very nature, leads to MCV architecture which is good for application like phometa.

8.2 Model-Controller-View Architecture

also talk about phometa modules hierarchy as well

8.3 Modes and Keymap

how modes work and interact with keymap

8.4 Examples of code — Pattern Matching

provide a full explanation of this part of code it is very interesting and small enough to show some aspect of functional programming

8.5 Compilation to Javascript, Html, and Css

8.6 Backend communication, Load / Save repository

8.7 Testing / Continious Integration

Chapter 9

Evaluation

9.1 Users Feedback — discuss with friends

On the 25^{th} of May 2016, it was the first day of project fair where students can demonstrate their work to other students and get a feedback so I went there and discuss about our projects. At this stage, the implementation is finished with Simple Arithmetic and Propositional Logic included in the standard library.

I started showing my project by explaining about phometa background and Simple Arithmetic using chapter 3 and 4 on this report. Then I asked them do to exercises on chapter 4 by having me as helper. All of them understood phometa and was able to proof a theorem. Finally, I asked them to try Propositional Logic, some of them really interest but of them didn't want to.

From my observation, all of them were comfortable to proof by clicking options from keymap pane rather than using keyboard shortcut. They also forgot to use searching pattern to select options faster.

There were a few parts of user interface that were not trivial enough, they needed to ask me what to do next, this should be fine if user have time to read whole tutorial.

On the bright side, most of them said that they really like the way that underlines was use to group sub-term rather than brackets (although they needed some time to familiar with it), they also said that the proof is quite easy to read and it will benefit newcomer.

There were several improvements that they suggest. Some of suggestions were easy to change (e.g. theorems should state its goal on header as well) so I changed it already. Some of other suggestions were quite big (e.g. make it mobile friendly and have a proper server) which can be considered as future works. We also managed to find some bugs[1] that I never found before, this gave me an opportunity to fix it in time.

[1] These bug are related Html and CSS rendering i.e. they are not related to phometa internal.

9.2 Users feedback — discuss with junior students

9.3 Professional Feedback

9.4 Strengths and Limitations

Chapter 10

Conclusion

10.1 Lesson Learnt

10.2 Future Works

10.2.1 Importation between module

10.2.2 Repository Verification on Loading

10.2.3 Adding new Formal Systems to Standard Library

Bibliography

[1] Evan Czaplicki et al. *Elm offical website.* `http://www.elm-lang.org`. [Online; accessed 3-February-2016].

[2] Encyclopedia Britannica. *Formal system.* `http://www.britannica.com/topic/formal-system`. [Online; accessed 29-January-2016].

[3] University of Cambridge and Technische Universität München. *Isabelle offical website.* `http://isabelle.in.tum.de/index.html`. [Online; accessed 15-May-2016].

[4] Chalmers and Gothenburg University. *Agda offical website.* `http://wiki.portal.chalmers.se/agda/pmwiki.php`. [Online; accessed 29-January-2016].

[5] Thierry Coquand. *Thierry Coquand homepage.* `http://www.cse.chalmers.se/~coquand`. [Online; accessed 29-January-2016].

[6] University of Edinburgh. *Proof General offical website.* `http://proofgeneral.inf.ed.ac.uk`. [Online; accessed 18-May-2016].

[7] *Homotopy Type Theory Repository.* `https://github.com/HoTT/HoTT`. [Online; accessed 18-May-2016].

[8] *Homotopy Type Theory Repository — Agda alternative.* `https://github.com/HoTT/HoTT-Agda`. [Online; accessed 18-May-2016].

[9] Inria. *Coq offical website.* `https://coq.inria.fr`. [Online; accessed 29-January-2016].

[10] Inria. *CoqIde offical website.* `https://coq.inria.fr/cocorico/CoqIde`. [Online; accessed 18-May-2016].

[11] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* New York, NY, USA: Oxford University Press, Inc., 1994. ISBN: 0-19-853835-9.

[12] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[13] Wikipedia. *Curry Howard correspondence.* `https://en.wikipedia.org/wiki/Curry-Howard_correspondence`. [Online; accessed 29-January-2016].

[14] Wikipedia. *Formal system.* `https://en.wikipedia.org/wiki/Formal_system`. [Online; accessed 29-January-2016].