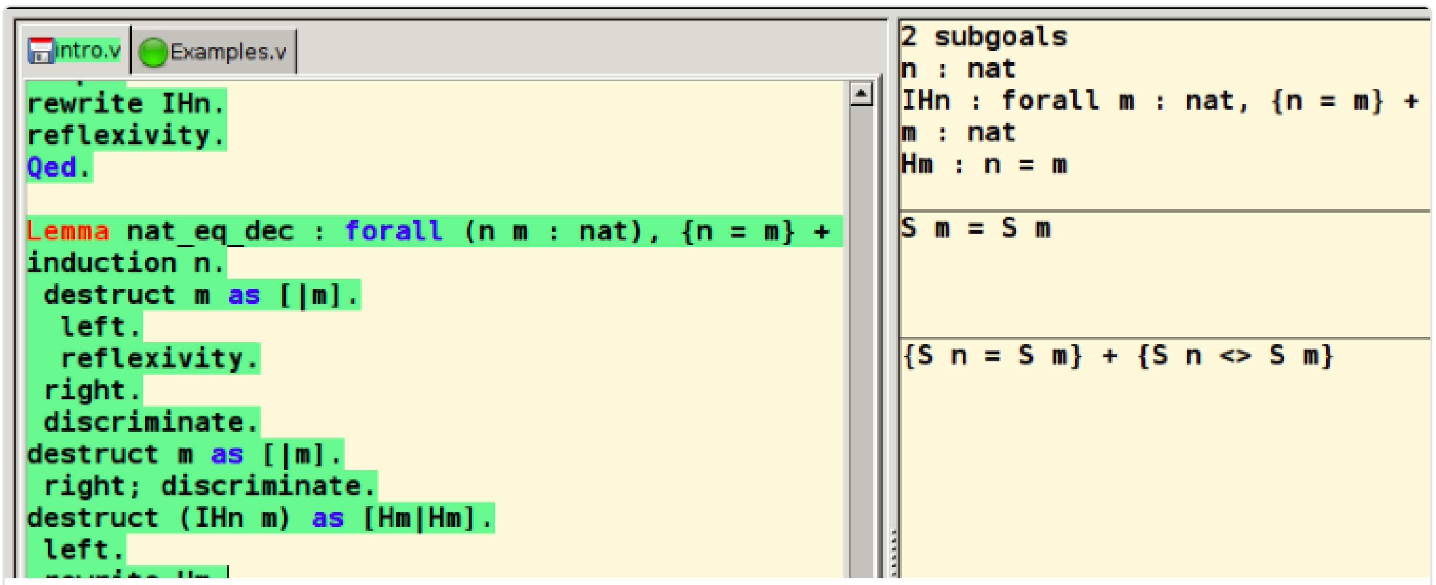
COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 2 My Mini Proof Assistant

Changelogs

- 2020-10-07 15:00
  Fix and add more description for the tasks.
- 2020-10-06 16:00
  Fix the bug in the demo program that using `destruct` will lead to crash.

End of Objective and Learning Outcomes



```
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), {n = m} +
induction n.
 destruct m as [|m].
  left.
  reflexivity.
 right.
 discriminate.
destruct m as [|m].
 right; discriminate.
destruct (IHn m) as [Hm|Hm].
 left.
```

```
2 subgoals
n : nat
IHn : forall m : nat, {n = m} +
m : nat
Hm : n = m

S m = S m

{S n = S m} + {S n <> S m}
```

Source: https://en.wikipedia.org/wiki/Proof_assistant

Did you know that computers can not only assist humans in computation, but also in proofs?

Introduction

In this programming assignment, we will be implementing a minimal proof assistant for propositional logic, using our knowledge of pointers, linked lists, and dynamic memory (de)allocation. With this tool, the users can develop a formal proof conveniently. Don't be afraid because this is more like an editor instead of an auto prover. And if you really cannot understand the formal proof, it is still possible to complete the tasks.

What is a proof assistant?

Proof assistants, also known as interactive theorem provers, are computer programs that assist humans in the development, verification and construction of large-scale rigorous proofs that are often too tedious to manually check by hand. They often provide the user with a much higher level of confidence in the correctness of his/her proofs compared to pen-and-paper arguments since every logical step of the proof, no matter how small, is checked by the machine right down to the axioms.

Propositional logic

Atoms

A proposition is something that we could attempt to provide a proof of. For example, the following statements are said to express propositions:

- The sun is made of plasma
- The moon is made of cheese
- 2 + 2 = 4
- 2 + 2 = 5
- The Riemann Hypothesis is true

- The Haskell programming language is type-safe (i.e. programs that compile can never encounter a runtime error) as long as no unsafe features such as `undefined` or unsafe coercions are used

And the following do not:

- What is your name?
- Remember to submit this programming assignment by the stipulated deadline
- Go clean the dishes!

Propositions that cannot be further decomposed from a purely logical perspective are called **atom**s. An atom (i.e. a propositional variable) is often represented by a single capital letter, e.g. ***P***.

Truth and Falsehood

Apart from atoms, it is often convenient to define two additional notions:

- The trivially true proposition, also known as a **tautology**, is written as $\top$
- Falsehood, also known as a **contradiction**, is written as $\bot$

Although they seems to be trivial, they are actually frequently used in our proofs.

Negation

We can also express the opposite side of a statement. If P is a proposition, then $\neg P$ is its negation.

Connectives

We can combine existing propositions (recursively) using **logical connectives** to create new propositions. For example:

- Conjunction (And): Given two propositions **P** and **Q**, $P \land Q$ is the proposition that both P and Q hold.
- Disjunction (Or): $P \lor Q$ is the proposition that *at least* one of P or Q hold (can be both).
- Material Implication (If): $P \to Q$ has the following properties.
  - Whenever P holds, Q must also hold.
  - When P does not hold, $P \to Q$ holds regardless of whether Q holds. We say in this case that the proposition $P \to Q$ is vacuously true, i.e. it is true but its truth does not tell us anything useful.

Note that in the treatment of formal logic, the two sides of a material implication (called the **antecedent** and **consequent** respectively) do **NOT** have causal relation. For example, the following are considered valid implications:

- If the Riemann Hypothesis is true then Haskell is type-safe
- If the moon is made of cheese then I am richer than Bill Gates
- If the sun is made of plasma then there are infinitely many primes

Other Notes

- The material implication $P \to Q$ is equivalent to $\neg P \lor Q$ (which can be proved by our system)
- Following this property, the negation $\neg P$ is equivalent to $P \to \bot$

Formal Proof

In a proof, we start from a set of **premises** (assumptions) and would like to derive a final **conclusion**. You may find that given a logical statement, we can always test the correctness by test all the T/F assignment for all the atoms. But in formal proof, we aim at proving just based on the form of the statements. Generally speaking, we make assumptions and derive new statements based on proved or assumed ones. And to derive new statements, we only use a fixed set of rules of inference.

A rule of inference is what we accept as a valid step in logical reasoning. It is important to note that different logical systems may have different rules of inference. For example, an inference rule called **modus ponens** works as follows (example taken from Wikipedia):

- If it's raining then it's cloudy.
- It's raining.
- *Therefore*, it's cloudy.

The first two bullet points are **premises**, i.e. what we know and/or assume to be true. The third bullet point (starting with "therefore") is the **conclusion**.

More generally, given any two propositions of the *form* ***P* → *Q*** and ***P***, we can **deduce *Q***. This rule of inference can also be written as follows in **inference rule notation**:

$$\frac{\begin{array}{l} A \to B \\ A \end{array}}{B}$$

Or, more succinctly: $P \to Q, P \vdash Q$ (the names of atoms are immaterial so we may freely interchange P, Q with A, B or vice versa.)

Some other common rules of inference include:

- Conjunction introduction: $p, q \vdash (p \land q)$
- Conjunction elimination (left): $(p \land q) \vdash p$. Similarly for the right.
- Disjunction introduction (left): $p \vdash (p \lor q)$. Similarly for the right.
- Disjunction elimination: $\{p \lor q, p \to r, q \to r\} \vdash r$

Note that this list is not meant to be exhaustive by any means. And the inference rules in our system are slightly different with these ones (which will be introduced later).

End of Introduction

Objective and Learning Outcomes

The objective of this assignment is to provide you practice on structures, pointers, linked lists and dynamic memory (de)allocation.

Upon completion of this assignment, you should be able to:

1. Use structures to group data together
2. Store, manage and manipulate data using pointers and linked lists
3. Manage memory properly in your program using `new` / `delete`
4. Construct a program towards the OOP style

End of Objective and Learning Outcomes

Using MMPA

From this point onwards, "My Mini Proof Assistant" will be abbreviated as MMPA.

In order to implement MMPA, you will need a basic understanding of how MMPA works, so here we go.

First of all, go to the Download section and download the demo program. Make sure you download the correct demo program for your operating system. Then execute the demo program to start following this brief tutorial on using MMPA.

Syntax

In MMPA, there are three things: *identifiers*, *propositional formulae* and *tactics*.

Identifiers

An identifier in MMPA is the name of a theorem or hypothesis (premise). An identifier is considered valid if all of the following conditions are met:

- It is at least one character long
- The first character is a lowercase English letter (`a-z`) or an underscore (`_`)
- Every other character is an English letter (`a-zA-Z`), digit (`0-9`) or underscore (`_`)

So the following are valid identifiers:

- `hyp`
- `hP`

Menu

- Introduction
- Objective and Learning Outcomes
- Using MMPA
- Tasks
- Resources & Sample I/O
- Submission & Grading
- FAQ

Page maintained by

Jingyang Zhang
Email: jzhangbs@connect.ust.hk
Last Modified: 10/07/2020 14:20:15

Homepage

Course Homepage

- `hQ`
- `h2_LKSJ298374__`
- `qQQQQQhahaAHAHAHA298374923`
- `_HP`

And the following are not:

- `HP`
- `HQ`
- `3433hsjdklfjsl`
- `H_A_H_A`

Propositional Formulae

Since it is difficult to type the special symbols found in propositional logic, we use their ASCII equivalents instead:

- Truth: `T`
- Falsehood: `F`
- Atom: any single uppercase English letter *except* `T` / `F`, e.g. `P`
- Conjunction: `/\`
- Disjunction: `\/`
- Implication: `->`
- Negation: `~`

Precedence and Associativity

`~` has highest precedence of all, followed by `/\`, then `\/` and finally `->`. So, for example, `~P /\ Q \/ R -> S` should be interpreted as `(((~P) /\ Q) \/ R) -> S`. Of couse, brackets override any and all operator precedence rules as usual.

Furthermore, even though `(P /\ Q) /\ R` and `P /\ (Q /\ R)` are logically equivalent and therefore we can often write `P /\ Q /\ R` unambiguously on paper, we still need to define an **associativity** for binary connectives for the sake of the computer. In MMPA, all binary connectives are *right*-associative, so `P /\ Q /\ R` is `P /\ (Q /\ R)`, and similarly for disjunction and implication.

Tactics

A tactic is a procedure that often corresponds to some logical step(s) of deduction on paper. In MMPA, the available tactics are different from the reference rules mentioned in the introduction. We will motivate their use through examples in the next subsection.

A note on the parser

Note that the parser for MMPA is *optimistic*, meaning that if you feed it malformed input, it may lead to weird behavior (or a program crash!).

In this programming assignment, all the parsing and pretty-printing have already been implemented for you so you do not have to worry about handling syntactically invalid input, though most certainly, you will be asked to handle other forms of invalid input (more on that later).

Modus Ponens

Let us prove Modus Ponens demonstrated in the introduction. In MMPA, we put all the premises and target conclusion into one statement: `premise_1 -> premise_2 -> ... -> premise_n -> conclusion` (think why they are equivalent). In this case, we have `(P -> Q) -> P -> Q`. In the main menu, we choose `P`, enter a name for this proof, and enter the statement.

```
1 subgoal(s) remaining.
Current focused subgoal:
__________________________________________________________________________
______
(P -> Q) -> P -> Q
```

Currently all the premises are encoded in the conclusion. We use `intro` tactic to extract them.

- `intro <name_of_new_hyp>`: If conclusion is of the form `P -> Q`, add `P` to the premise and change the conclusion to `Q`

```
> intro hPQ
> intro hP
```

which leaves us with the following proof state:

```
1 subgoal(s) remaining.
Current focused subgoal:
hP : P
hPQ : P -> Q
________________________________________________________________________
______
Q
```

Now, you would probably like to perform **forward reasoning** as follows: "From `P` and `P -> Q`, deduce `Q` and we're done."

But, at least in the realm of proof assistants, it is often more natural to perform **backwards reasoning** instead. That is, instead of deducing `Q` from `P` and `P -> Q`, we say, "Okay, we want to prove `Q` and we know that `P -> Q`. So it *suffices* to prove `P` instead, which holds by assumption and we're done." In other words, we keep finding the prerequisites for proving the conclusion until the prerequisites are the ones we have. If there are multiple prerequisites, we will have multiple subgoals to prove.

Now we use `apply` tactic.

- `apply <name_of_exist_hyp>`: If we have a premise `P -> Q` and the conclusion is exactly `Q`, change the conclusion to `P`; If the premise is `Q`, the (sub)proof is finished; If the premise is `P1 -> P2 -> ... -> Pn -> Q`, create `n` subproofs where the `i`-th subproof has conclusion `Pi`

```
> apply hPQ
1 subgoal(s) remaining.
Current focused subgoal:
hP : P
hPQ : P -> Q
________________________________________________________________________
______
P
Enter a tactic:
> apply hP
No more subgoals.
Theorem added to existing results
```

Note that `apply` used in this way is exactly modus ponens (and therefore a valid logical deduction step), but in a backwards fashion.

De Morgan's Law

We will prove the De Morgan's Law `~(P /\ Q) -> ~P \/ ~Q` as the second example.

```
h1 : ~(P /\ Q)
________________________________________________________________________
______
~P \/ ~Q
```

Similarly, let's see what can we do with the conclusion. Since this is a disjunction, if we can derive `~P`, then `~P \/ <something_else>` must hold (which refer to disjunction introduction). To do this, let's try `left` tactic.

- `left (right)`: if the conclusion is a disjunction, change the conclusion to the left (right) part of the original one.

```
> left
1 subgoal(s) remaining.
Current focused subgoal:
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P
```

But this is not correct. If P is true and Q is false, we will have true premises but false conclusion. So this tactic is not appropriate.

If there is no breakthrough from the the existing statements, we consider to introduce new premises. Here we introduce the `excluded_middle` tactic. The intuition is that between a statement `P` and its negation `~P`, exactly one of them is true. If we consider two cases and can derive the conclusion by assume `P` and `~P` respectively, we can prove the conclusion.

- `excluded_middle <name_of_new_hyp> : <statement>`: create two subgoals, and add the input statement and its negation to the premises respectively.
- `rotate <n>`: switch to the `n`-th subgoal next to the current one.

```
> excluded_middle h2 : P
2 subgoal(s) remaining.
Current focused subgoal:
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
Enter a tactic:
> rotate 1
2 subgoal(s) remaining.
Current focused subgoal:
h2 : ~P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
```

The second subgoal is provable by `left`. For the first one, we similarly use `excluded_middle h3 : Q` and easily prove the subgoal with `~Q`. Now we only have one subgoal.

```
1 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
```

We find that the premises cannot be all true, which means this cases is impossible. So we just need to show that the premises lead to a contradiction which then can imply everything (by the definition of material implication). Now we use `exfalso` tactic.

- `exfalso`: change the conclusion to `F`.

```
> exfalso
1 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________

______
F
```

Recall that `~(P /\ Q)` is equivalent to `(P /\ Q) -> F`. So we can use the `apply` tactic.

```
> apply h1
1 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________

______
P /\ Q
```

To prove this conjuction, we need to show that both sides hold. Luckily we have them in the premises. Now we use `split` tactic to set the new goal.

- `split`: if the conclusion is a conjunction, create 2 subgoals and set the conclusions as the two sides of the conjunction respectively.

```
> split
2 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________

______
P
Enter a tactic:
> rotate 1
2 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________

______
Q
```

Both subgoals trivially hold.

Available Tactics

Here we show all the available tactics in MMPA (including the ones in the previous section).

Constructive Logic

- `trivial`: if the conclusion is `T`, finish the subgoal.
- `contradiction`: if the premises include `F`, finish the subgoal.
- `exfalso`: change the conclusion to `F`.
- `intro <name_of_new_hyp>`: If conclusion is of the form `P -> Q`, add `P` to the premise and change the conclusion to `Q`
- `apply <name_of_exist_hyp>`:
  - If we have a premise (*or proven theorem*) `P -> Q` and the conclusion is exactly `Q`, change the conclusion to `P`.
  - If the premise is `Q` and the conclusion is exactly `Q`, the (sub)proof is finished.
  - If the premise is `P1 -> P2 -> ... -> Pn -> Q` and the conclusion is exactly `Q`, create `n` subproofs where the `i`-th subproof has conclusion `Pi`.
- `destruct <name_of_exist_hyp> as <name_of_new_hyp_1> <name_of_new_hyp_2>`:

    - if the premise (*or proven theorem*) `<name_of_exist_hyp>` is a conjunction `P /\ Q`, add `P` and `Q` to the premises.
    - if the premise (*or proven theorem*) `<name_of_exist_hyp>` is a disjunction `P \/ Q`, create 2 subgoals and add `P` and `Q` respectively to them (one subgoal has `P` and the other one subgoal has `Q`).
- `split`: if the conclusion is a conjunction `P /\ Q`, create 2 subgoals and set the conclusions as the two sides of the conjunction respectively (one subgoal has conclusion `P` and the other one has `Q`).
- `left (right)`: if the conclusion is a disjunction, change the conclusion to the left (right) part of the original one.

Classical Logic

- `excluded_middle <name_of_new_hyp> : <statement>`: create two subgoals, and add the input statement and its negation to the premises respectively (if input `P`, one subgoal has `P` and the other one has `~P`).
- `by_contradiction <name_of_new_hyp>`: add the negation of the current conclusion to the premises and change the conclusion to `F`.

Proof Management

- `rotate <n>`: switch to the `n`-th subgoal after the current one.
- `clear <name_of_exist_hyp>`: remove the premise.
- `abort`: give up the current proof and go back to the main menu.

More Exercises

Here we provide more exercises for you if you want to get more familiar with the proving system.

1. Prove that conjunction is commutative: `P /\ Q -> Q /\ P`. Solution
2. Prove that disjunction is commutative: `P \/ Q -> Q \/ P`. Solution
3. Prove that conjunction is associative: `P /\ Q /\ R -> (P /\ Q) /\ R`; `(P /\ Q) /\ R -> P /\ Q /\ R`. Solution
4. Prove that disjunction is associative: `P \/ Q \/ R -> (P \/ Q) \/ R`; `(P \/ Q) \/ R -> P \/ Q \/ R`. Solution
5. (Extended, no solution provided) Prove that conjunction distributes over disjunction, i.e. `P /\ (Q \/ R) -> P /\ Q \/ P /\ R` and its **converse** (the reverse implication).
6. (Extended, no solution provided) Prove that disjunction also distributes over conjunction, but first of all, what would the statement(s) even look like?

End of Using MMPA

Tasks

Code Structure

Propositional Formulae

In MMPA, each propositional formula is represented as an abstract syntax tree (AST):

```
struct Formula {
  // The type of this formula
  // A propositional formula is either a constant (T/F), atom (P/Q/R/etc.) or
  // (/\ / \/ / ->)
  enum Type { CONSTANT, ATOM, CONNECTIVE };
  Type type;

  // Name/symbol of this part of the formula
  // This is at most 2 characters in length, excluding the trailing NUL byte
  char name[3];

  // Subformulae of this formula, if any (e.g. the subformulae of P -> Q are
  // These are encoded as left and right subtrees of the corresponding tree i
  // For logical constants and atoms, both fields should be nullptr; otherwis
  // P -> Q, the left subtree is P and the right subtree is Q
  Formula *left;
  Formula *right;
};
```

You do **not** need to understand this structure in order to complete the programming assignment (though you are more than welcome to, if you are interested in compiler and/or programming language theory); suitable helper functions have been provided for you which should allow you to deal with them abstractly as propositions.

List of proven theorems / list of premises

The list of proven theorems, as well as the list of premises in each subgoal are represented via the following structure:

```
struct Theorem {
  // The name of our theorem
  char *name;
  // The statement of our theorem
  Formula *stmt;
  // Other theorems
  Theorem *next;
};
```

This is just a **singly linked list** with each node storing the name of the theorem / hypothesis, the corresponding statement, and a pointer to the next theorem / hypothesis (or `nullptr` if the given theorem / hypothesis is the last one). Note in particular that the `name` field is expected to be dynamically allocated (or `nullptr`).

Subgoals and Proof State

Each subgoal is represented by the following structure:

```
struct Context {
  // List of local facts
  Theorem *premises;
  // Desired conclusion
  Formula *conclusion;
  // Previous/next subgoal
  Context *prev;
  Context *next;
};
```

The `premises` data member is the list of hypotheses above the line in the given subgoal, and `conclusion` is the conclusion below the line which we're trying to prove. `prev` and `next` are pointers to the previous and next subgoals in the current proof respectively, with the `prev` pointer of the first subgoal pointing towards the last subgoal (i.e. we have a circular doubly linked list of subgoals).

The proof state is then given by the following structure:

```
struct ProofState {
  // Current focused subgoal
  Context *focus;
};
```

The `focus` data member should point to the current focused subgoal, which is *always* the first subgoal in our circular doubly linked list (or `nullptr` if there are no subgoals remaining).

Utility Functions

As hinted earlier, the parsing and pretty-printing of propositional formulae, tactics, etc., have already been implemented for you and you do not need to (and in fact should not) modify them in any way. The functions are declared in `formula.h` and are listed as follows.

```
// Pretty-printer for propositional formulae
void print_formula(const Formula *);

// Parser for propositional formulae
Formula *parse_formula(char *);

// Identifier validator
bool is_valid_id(const char *);
```

Additionally, the following helper functions for manipulating `Formula`e are provided so you should never need to manipulate the AST directly:

- `void delete_formula(Formula *formula)`: Recursively deallocates memory previously allocated to `formula`
- `Formula *formula_deep_copy(const Formula *formula)`: Returns a deep copy of `formula`
- `bool formulae_equal(const Formula *formula1, const Formula *formula2)`: Checks whether `formula1` and `formula2` are syntactically equal
- `bool formula_is_T(const Formula *formula)`: Returns whether `formula == T`
- `Formula *formula_F()`: Allocates and returns a `Formula` representing bottom
- `bool formula_is_disj(const Formula *formula)`: Returns whether the proposition is a disjunction
- `Formula *&formula_disj_left(Formula *formula)`: Returns a reference to the left side of a disjunction, or exits the program with an error otherwise
- `Formula *&formula_disj_right(Formula *formula)`: Returns a reference to the right side of a disjunction, or exits the program with an error otherwise
- `bool formula_is_impl(const Formula *formula)`: Returns whether the proposition is an implication
- `Formula *&formula_impl_antecedent(Formula *formula)`: Returns a reference to the antecedent of an implication, or exits the program with an error otherwise
- `Formula *&formula_impl_consequent(Formula *formula)`: Returns a reference to the consequent of an implication, or exits the program with an error otherwise
- `bool formula_is_conj(const Formula *formula)`: Returns whether the proposition is a conjunction
- `Formula *&formula_conj_left(Formula *formula)`: Returns a reference to the left side of a conjunction, or exits the program with an error otherwise
- `Formula *&formula_conj_right(Formula *formula)`: Returns a reference to the right side of a conjunction, or exits the program with an error otherwise
- `bool formula_is_F(const Formula *formula)`: Returns whether `formula == F`
- `Formula *formula_neg_of(Formula *formula)`: Given `P`, allocates and returns `~P`. Keep in mind that `P` is *not* copied in the AST representation of `~P`.

The functions for printing the premises, theorems and subgoals are also given. They may be helpful for you to implement other functions.

```
void print_thms(const Theorem *thms)
```

Given a list of hypotheses / proven theorems `thms`, print them to the console. Each theorem / hypothesis should be displayed on a separate line, in the form `hyp : P` where `hyp` is the theorem/hypothesis name and `P` its statement.

```
void display_focus(const ProofState &proof_state)
```

Display the current focused subgoal onto the console, i.e. print the list of hypotheses, followed by a horizontal rule (consisting of *exactly* 80 underscores), followed by the conclusion to be proven.

In the functions to be completed, you may also need to print some messages. Consult the demo program and compare your output with it in case of doubt. **Note in particular that your output must match EXACTLY that of the demo program, INCLUDING all whitespace and newlines.**

Task 1: Preliminaries (0%)

Implement each of the functions described below. Note that **even though there is no explicit allocation of marks for this task, these functions are essential for the basic functioning of MMPA so implementing them incorrectly may severely impact your marks in subsequent tasks**. Anyway, most of these functions should be straightforward to implement and should serve as a warm-up for subsequent tasks.

```
void delete_thms(Theorem *thms)
```

Recursively deallocates the memory allocated to the given linked list of theorems / hypotheses. You need to deallocate **all** the elements in the linked list. *Remember in particular that the `name` field is also assumed to be dynamically allocated (or `nullptr`) so you will have to delete it as well (and make sure you use the correct variant of delete ;-).*

```
const Theorem *id_exists(const char *id, const Theorem *thms)
```

Returns the pointer to a theorem / hypothesis with name `id` if it is present in `thms` and `nullptr` otherwise.

```
void init_proof(ProofState &proof_state, Formula *stmt)
```

Initialize the given `proof_state` with exactly 1 subgoal with no premises and `stmt` as the conclusion to be proven. You may assume that `stmt` has already been dynamically allocated for you; therefore, you do *not* need to perform a deep copy of it - just use the given pointer.

```
bool has_subgoals(const ProofState &proof_state)
```

Returns `true` if the given state contains at least 1 subgoal and `false` otherwise.

```
void add_thm_to_database(Theorem *&thms, const char *name, Formula *stmt)
```

Add a theorem / hypothesis with name `name` and statement `stmt` to the head of the given list `thms`. Note that parameter `name` is statically allocated while the `name` field of a theorem / hypothesis should be dynamically allocated so you will have to dynamically allocate a new buffer and copy over the contents of `name` to your theorem / hypothesis. On the other hand, you may assume that `stmt` has been dynamically allocated for you already.

```
int num_subgoals(const ProofState &proof_state)
```

Returns the number of subgoals in the current proof state.

```
void clear_proof_state(ProofState &proof_state)
```

Clears the given proof state by removing (and deallocating) all subgoals, then setting the `focus` to `nullptr`.

Task 2: Tactics for constructive logic, Part I (24%)

Preliminaries aside, let us now implement some of the simpler tactics related to constructive logic in MMPA. You may wish to refer to the Tactic Reference for more details if necessary.

```
void tactic_trivial(ProofState &proof_state)
```

Implement the `trivial` tactic. The current proof state is provided as argument. In addition, you need to handle the following invalid inputs:

- Do nothing if `focus` is `nullptr`
- If the tactic is applied where the conclusion to be proven is not `T`, print the message `Tactic 'trivial' failed: conclusion is not 'T'`.

```
void tactic_exfalso(ProofState &proof_state)
```

Implement the `exfalso` tactic. In addition, you should handle the following invalid input:

- Do nothing if `focus` is `nullptr`

```
void tactic_left(ProofState &proof_state)
```

Implement the `left` tactic. In addition, you should handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If the tactic is applied to a conclusion that is not a disjunction, print the message `Tactic 'left' failed: conclusion is not a disjunction`.

```
void tactic_right(ProofState &proof_state)
```

Implement the `right` tactic. In addition, you should handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If the tactic is applied to a conclusion that is not a disjunction, print the message `Tactic 'right' failed: conclusion is not a disjunction`.

```
void tactic_intro(ProofState &proof_state, const char *hyp)
```

Implement the `intro` tactic; `hyp` is the name of the hypothesis, e.g. as in `intro hyp`. Note again that `hyp` is statically allocated so you'll need to make a copy. In addition, you should handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If the tactic is applied to a conclusion that is not an implication, print the message `Tactic 'intro' failed: conclusion is not an implication`.

- If the hypothesis name is already taken, print the message `Tactic 'intro' failed: hypothesis name is already taken`.

```
void tactic_split(ProofState &proof_state)
```

Implement the `split` tactic. In addition, handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If the tactic is applied to a conclusion that is not a conjunction, print the message `Tactic 'split' failed: conclusion is not a conjunction`.

Task 3: Tactics for constructive logic, Part II (28%)

Implement 3 more tactics related to constructive logic. The marks for this task are calculated separately from Task 2 since these tactics (except `contradiction`) are rather tricky to implement.

```
void tactic_contradiction(ProofState &proof_state, const Theorem *proven_resu
```

Implement the `contradiction` tactic. `proven_results` is the list of currently proven theorems. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If there is no hypothesis or proven theorem of "type" `F`, print the message `Tactic 'contradiction' failed: no contradiction found in local or global context`.

```
void tactic_destruct(ProofState &proof_state, const Theorem *proven_results,
```

Implement the `destruct` tactic. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If `hyp` is not the name of an existing theorem / hypothesis, print the message `Tactic 'destruct' failed: given hypothesis name not found in local or global context`.
- If `hyp` is neither a conjunction nor disjunction, print the message `Tactic 'destruct' failed: the given hypothesis is not a conjunction or disjunction`.
- If either of `hyp1` or `hyp2` are already taken names by hypotheses in the local context, print the message `Tactic 'destruct' failed: hypothesis name is already taken`.
- If we are destructing over a conjunction but `hyp1` and `hyp2` are the same name, print the message `Tactic 'destruct' failed: the two hypothesis names must be unique`. **This restriction does not apply to disjunctions.**

Furthermore, it is EXTREMELY important (for the purposes of grading) that you get the order of hypotheses right when adding them to the local context (i.e. they must match the demo program EXACTLY). For example, given the following context:

```
hAB : A /\ B
hC : C
hD : D
__________________________________________________________________________

______
E
```

`destruct hAB as hA hB` should give the following new context:

```
hA : A
hB : B
hAB : A /\ B
hC : C
hD : D
___________________________________________________________________________
______
E
```

```
void tactic_apply(ProofState &proof_state, const Theorem *proven_results, con
```

Implement the `apply` tactic. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If `hyp` is not the name of any theorem / hypothesis, print the message `Tactic 'apply' failed: hypothesis not found in local or global context`.
- If the consequent of the hypothesis does not match the conclusion we are trying to prove, print the message `Tactic 'apply' failed: hypothesis is not applicable to conclusion`.

Task 4: Tactics for classical logic (14%)

If you managed to complete Task 3 then this task (and subsequent tasks) should be easy for you ;-)

```
void tactic_excluded_middle(ProofState &proof_state, const char *hyp, Formula
```

Implement `excluded_middle`. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If the hypothesis name is already taken, print the message `Tactic 'excluded_middle' failed: hypothesis name is already taken`.

```
void tactic_by_contradiction(ProofState &proof_state, const char *hyp)
```

Implement `by_contradiction`. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If the hypothesis name is already taken, print the message `Tactic 'by_contradiction' failed: hypothesis name is already taken`.

Task 5: Tactics for proof management (14%)

```
void tactic_rotate(ProofState &proof_state, int n)
```

Implement `rotate`. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`

```
void tactic_clear(ProofState &proof_state, const char *hyp)
```

Implement `clear`. Handle the following invalid input:

- Do nothing if `focus` is `nullptr`
- If `hyp` is not the name of an existing hypothesis, print the message `Tactic 'clear' failed: no hypothesis of the given name found`.

Memory Management (20%)

For each test case, there will be additional score for memory management. Once you implement all of the tasks above correctly, make sure to manage your memory properly since it accounts for 20% of your assignment grade.

End of Description

Resources & Sample I/O

- Skeleton code: pa2_skeleton.zip
- Makefile
  - Windows: Makefile
  - Mac/Linux: Makefile
- Demo executables:
  - Windows: pa2_demo.exe
  - macOS: pa2_demo
  - Linux: pa2_demo

  Note for macOS: the executable will likely be blocked from executing. If you trust the executable, go to "System Preferences > Security & Privacy > General" to unblock it.

Run demo program on CS lab 2 machines

In case you cannot run demo program (on Mac), You can SSH to the CS lab2 machines and try the demo program for Linux.

```
ssh <CSD_username>@csl2wk01.cse.ust.hk    #'wk01' can be wk01-wk53
wget
https://course.cse.ust.hk/comp2012h/assignments/pa2/files/linux/pa2_demo -
-http-user=<CSD_username> --http-password=<CSD_password>
chmod +x pa2_demo
./pa2_demo
```

Sample I/O

Your program should produce the following output. You may wish to compare your program output against the expected output using a diff checker.

```
Welcome!
You are using My Mini Proof Assistant (TM), v0.1.0beta
Choose an option:
P. Prove a theorem
B. Browse existing theorems
Q. Quit
> P
Enter the name of the theorem to be proven:
> demorgan
Enter the statement of the theorem to be proven:
> ~(P /\ Q) -> ~P \/ ~Q
1 subgoal(s) remaining.
Current focused subgoal:
__________________________________________________________________________
______
~(P /\ Q) -> ~P \/ ~Q
Enter a tactic:
> intro h1
1 subgoal(s) remaining.
Current focused subgoal:
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
Enter a tactic:
> excluded_middle h2 : P
2 subgoal(s) remaining.
Current focused subgoal:
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
Enter a tactic:
> excluded_middle h3 : Q
3 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
Enter a tactic:
> exfalso
3 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
F
Enter a tactic:
> apply h1
3 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
P /\ Q
Enter a tactic:
> split
4 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
```

```
______
P
Enter a tactic:
> apply h2
3 subgoal(s) remaining.
Current focused subgoal:
h3 : Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
Q
Enter a tactic:
> apply h3
2 subgoal(s) remaining.
Current focused subgoal:
h3 : ~Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
Enter a tactic:
> right
2 subgoal(s) remaining.
Current focused subgoal:
h3 : ~Q
h2 : P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~Q
Enter a tactic:
> apply h3
1 subgoal(s) remaining.
Current focused subgoal:
h2 : ~P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P \/ ~Q
Enter a tactic:
> left
1 subgoal(s) remaining.
Current focused subgoal:
h2 : ~P
h1 : ~(P /\ Q)
__________________________________________________________________________
______
~P
Enter a tactic:
> apply h2
No more subgoals.
Theorem added to existing results
Choose an option:
P. Prove a theorem
B. Browse existing theorems
Q. Quit
> b
demorgan : ~(P /\ Q) -> ~P \/ ~Q
Choose an option:
P. Prove a theorem
B. Browse existing theorems
Q. Quit
> q
See you next time!
```

End of Download

Submission & Grading

Deadline: 19 October 2020 Friday HKT 23:59.
You may earn 8% course grade for each PA via Automated Grading on the ZINC Online Submission System. Please zip `mmpa.cpp` as `pa2.zip` for submission to ZINC.

We will have multiple standard I/O tests to grade the submission. Apart from them, we will also provide sample test cases for you to validate your code.

Please note that the samples do not show all possible cases. It is part of the assessment for you to design your own test cases to test your program. Please also remember to remove or comment out any debugging message(s) that you might have added, before submitting your code.

End of Submission & Grading

FAQ

Frequently Asked Questions

This section will be expanded as your fellow classmates ask more questions about this programming assignment. It is in your interest to **check this section regularly for updates and clarifications** *even if you have already submitted your work then*.

This section is currently empty.

End of FAQ