Practical 3 — Propositional Logic

Albert Rizaldi

November 23, 2015

1 Preliminaries

The purpose of this programming exercise is to show the computational aspect of propositional logic materials covered in the exercise. This exercise requires you to program two modules: *parser* and *entailment*. The parser module reads ASCII strings and converts them into propositional logic formula. The entailment module reads two parsed propositional logic formulae α and β, and then it decides whether $\alpha \models \beta$ or not. We then use these two modules for solving the problems in the exercise sheet.

Programming language. You are free to use any programming language except OCAML —sorry for OCAML programmer. The reason is because this programming exercise is based on the work by Harrison [1], where he used OCAML for the code. However, it is advised to use functional programming language for this exercise. The only reason is because it is much easier to translate code from OCAML to another functional programming languages. However, if you are not comfortable with them —and you are not willing to invest time to learn them— choose programming language you are most familiar with.

Main reading. You are also strongly advised to read Harrison's book [1]. You can look for the book in the computer science branch library. Additionally, you can also access the book through SAFARI online book website. As a TUM student, you can access all online books available in that website. You can access SAFARI online book website from this URL:

`www.ub.tum.de/en/news/safari-books-online`.

You only need to read from Chapter 1 to Chapter 2.3 for this programming exercise.

Miscellaneous. You are strongly advised to collaborate with other students over the general idea about how to solve the exercise. However, you must submit individually. Plagiarising other people's code will cause you to fail this course. If you take the code from the website, give credit where it's due. Write in comments where the code is taken from. The deadline for this exercise is on Sunday 10$^{\text{th}}$ January 2016 at 23:59.

2 Parser

Main reference : Harrison's book Chapter 1.5–1.7 and Chapter 2.1

Your task is to build a parser for reading ASCII and obtain its representation in Abstract Syntax Tree (AST). We use the same ASCII representation for propositional logic connectives and constants as in Chapter 2.1 ([1]). The parser implementation can be divided into two parts: *lexer* and actual *parser*.

2.1 Lexer

A basic implementation of a lexer is shown in Chapter 1.7. Implement the function `lex` with the same tokens (`space`, `punctuation`, `symbolic`, `numeric`, `alphanumeric`) as in the book.

2.2 Actual parser

Start with defining the new recursive datatype `Formula` for propositional logic formula in your programming language of choice (see Chapter 2.1).

Caveats. Although Harrison included `Forall` and `Exists` in the datatype definition, you can safely ignore them. Harrison's definition also used type variable of $'a$. This type variable represents the type of the propositional variable (see the constructor `Atom`). However, you can ignore the type variable by fixing the type of propositional variable with type *String* or *Int* in your programming language of choice.

Make sure you differentiate between *True* and *False* in propositional logic and `True` and `False` which is of type Boolean in your programming language of choice.

The order of precedence for the connectives are arranged in increasing order as $\Leftrightarrow, \Rightarrow, \vee, \wedge, \neg$. By using this information, try to devise the grammar for propositional logic with *Recursive Descent Parsing* approach (see Chapter 1.7). With this grammar, you can readily adopt the approach used in the book to implement your parser.

3 Entailment

Main reference : Harrison's book Chapter 2.1–2.3

Semantics of Propositional Logic. From the previous exercise, you should have a firm understanding of propositional logic syntax. The second important part of propositional logic is its semantics. For propositional logic, the semantics is very easy. The semantics is based on the notion of *valuation*, a function which assign each variable to boolean value of either *true* or *false*. Depends on your choice of datatype for representing propositional variable ($'a$), the *valuation* has the type

$$\texttt{valuation} \;::\; 'a \longrightarrow \texttt{Bool}$$

where `Bool` is the type of boolean in your programming language of choice. Implement the function `eval` which has the following type

$$\texttt{eval} \;::\; \texttt{Formula} \longrightarrow ('a \longrightarrow \texttt{Bool}) \longrightarrow \texttt{Bool}$$

Function `eval fm v` will decide whether formula `fm` is true or not under the valuation `v`.

Function `atoms`. One of the function required in this programming exercise is the function `atoms` which collects all propositional variables in a formula. The recursive definition of this function is given in Chapter 2.1. Implement this function by representing the collection of all propositional variables as set. This will avoid the problem of duplicated propositional variables which will happen if you use other datatype such as list.

Function `onallvaluations`. Another function required in this programming exercise is the function which tests whether the current formula evaluates to true on all valuations. This function clearly requires the function `atoms` defined previously. From this collection of proposition variables, implement this function by substituting each of them with the value of *True* and *False*, and checking whether each combination evaluates to `True`. For propositional formula with n propositional variables, there are 2^n possible valuations.

Tips. Remember that it has to evaluate to `True` on **all** valuations. If there is a valuation which results to `False`, there is no point on continuing with another valuation.

With all of these modules, you are ready to implement the following functions:

- `tautology`[1] :: `Formula` $\longrightarrow$ `Bool`
- `unsatisfiable` :: `Formula` $\longrightarrow$ `Bool`
- `satisfiable` :: `Formula` $\longrightarrow$ `Bool`
- `entails` :: `Formula` $\longrightarrow$ `Formula` $\longrightarrow$ `Bool`

The first three functions are covered in Harrison's book. For function `entail`, we can use the following (meta-)theorem:

Theorem 3.1 *For any two propositional formula α and β, $\alpha \models \beta$ if and only if the formula $\alpha \land \neg\beta$ is unsatisfiable.*

Theorem 3.2 *For any two propositional formula α and β, $\alpha \models \beta$ if and only if the formula $\alpha \Rightarrow \beta$ is valid (a tautology).*

You may use any of these two theorems to implement the last function.

4 Exercise

In this part, we will use the functions implemented in previous sections to answer the questions in exercise.

Problem 4.1. Use the function `entails` to check whether the following entailment is true or not.

1. *False* $\models$ *True*
2. *True* $\models$ *False*
3. $(A \land B) \models (A \Leftrightarrow B)$
4. $(A \Leftrightarrow B) \models A \lor B$
5. $(A \Leftrightarrow B) \models \neg A \lor B$

Question: Compare the output from your function and the result you obtain by using pencil-and-paper derivation. How is the function differs from the approach we use in pencil-and-paper derivation?

[1]This is the same term with validity.

Problem 4.2.2 Use the function `tautology, satisfiable, unsatisfiable` to check whether the following formulae is tautology, satisfiable, or unsatisfiable. Compare the output with the result from your pencil-and-paper derivation.

1. $\mathit{Smoke} \Rightarrow \mathit{Smoke}$
2. $(\mathit{Smoke} \Rightarrow \mathit{Fire}) \Rightarrow (\neg \mathit{Smoke} \Rightarrow \neg \mathit{Fire})$
3. $\mathit{Smoke} \vee \mathit{Fire} \vee \neg \mathit{Fire}$
4. $(\mathit{Fire} \Rightarrow \mathit{Smoke}) \wedge \mathit{Fire} \wedge \neg \mathit{Smoke}$

Problem 4.2.3 Represent what B says with your parser.

```
bsays = parse "b <=> (a <=> ~a)"
```

where `parse` is your parser implementation. Do the same with what C says.

```
csays = parse "c <=> ~b"
```

Construct a knowledge base —`kb` of type Formula— by performing conjunction of what B and C says. By using the function `entails`, check whether the knowledge base entails whether A is a knight. Check also whether it entails whether A is a knave. Perform these checks for B and C as well.

Questions. Check your answer about what B and C are with the answer from pencil-and-paper derivation in exercise. Can you decide what A is? Is it true that if $KB \not\models \alpha$, then it must be that $KB \models \neg\alpha$? Explain your answers.

References

[1] HARRISON, J. *Handbook of practical logic and automated reasoning*. 2009.