Chapter 1:

Propositional Logic Syntax

In this chapter we present a formal **syntax** for describing logical propositions. Consider the following example of a natural-language sentence that has some logical structure: "If it rains on Monday then we will either hand out umbrellas or hire a bus." This sentence talks about three basic **propositions**, each of which may potentially be either true or false: p1="it rains on Monday", p2="we will hand out umbrellas", and p3="we will hire a bus". These three propositions are connected logically as follows: "p1 implies (p2 or p3)", which we will write as '(p1→(p2|p3))'.

Our goal in this chapter is to formally define a language to capture these types of sentences. The motivation for defining this language is that it will allow us to precisely and formally analyze their *meaning*. For example, we should be able to formally deduce from this sentence that if it we neither handed out umbrellas nor hired a bus, then it did not rain on Monday. We purposefully postpone to the next chapter such a discussion of **semantics**, i.e., of the meaning that we assign to sentences in our language, and focus in this chapter only on the **syntax**, i.e., on the rules of grammar for forming sentences.

1 Propositional Formulas

Our language is called **Propositional Logic**, and while there are various variants of the exact rules of this language (allowing for various logical connectives or for various rules about whether and when parentheses may be dropped), the exact variant used is not very important, but rather the whole point is to fix a single specific set of rules and stick with it. Essentially everything that we say about this specific variant will hold with only very minor modifications for other variants as well. So here is the formal definition with which we will stick:

Definition (Propositional Formula)**.** The following strings are valid **propositional formulas**:

- An **atomic proposition**: a letter in 'p'. . . 'z', optionally followed by a sequence of digits.

  Examples: 'p', 'y12', 'z035'.
- 'T'.
- 'F'.
- '~ϕ', where ϕ is a valid propositional formula.
- '(ϕ&ψ)' where each of ϕ and ψ is a valid propositional formula.
- '(ϕ|ψ)' where each of ϕ and ψ is a valid propositional formula.

- '$(\phi \rightarrow \psi)$' where each of ϕ and ψ is a valid propositional formula.

These are the only valid propositional formulas. For example, '~((~x&(p007|x))→F)' is a valid propositional formula.

This definition concerns rules for forming *strings*, that is, finite sequences of characters, and specifies which string is a valid propositional formula and which is not. (Again, we have deliberately not yet assigned any meaning to such strings, but the reader will surely guess that 'T' and 'F' stand for *True* and *False*, respectively, and that '~', '&', '|', and '→' stand for *Not*, *And*, *Or*, and *Implies*, respectively.) We remark that in many logic textbooks, the symbol '$\neg$' is used instead of '~', the symbol '$\wedge$' is used instead of '&', and the symbol '$\vee$' is used instead of '|'.

Our choice of symbols in this book was indeed influenced by which symbols are easy to type on a computer. For your convenience, the file `propositions/syntax.py` defines functions for identifying strings that contain the various **tokens**, or basic building blocks, allowed in propositional formulas.[1] The symbol '→' is not a standard character, so in Python code we will represent it as the two-character sequence `'->'`.

propositions/syntax.py

```
def is_variable(string: str) -> bool:
    """Checks if the given string is an atomic proposition.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is an atomic proposition, ``False``
        otherwise.
    """
    return string[0] >= 'p' and string[0] <= 'z' and \
           (len(string) == 1 or string[1:].isdigit())

def is_constant(string: str) -> bool:
    """Checks if the given string is a constant.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a constant, ``False`` otherwise.
    """
    return string == 'T' or string == 'F'

def is_unary(string: str) -> bool:
    """Checks if the given string is a unary operator.

    Parameters:
```

[1]The decorator that precedes the definition of each of these functions in the code that you are given **memoizes** the function, so that if any of these functions is called more than once with the same argument, the previous return-value for that argument is simply returned again instead of being recalculated. This has no effect on code correctness since running these functions has no side effects, and their return values depend only on their arguments, but this does speed-up the execution of your code. It may seem silly to perform such optimizations with such short functions, but this will in fact dramatically speed-up your code in later chapters, when such functions will be called many many times from within various recursions. We use this decorator throughout the code that you are given in various places where there are speed improvements to be gained.

```
        string: string to check.

    Returns:
        ``True`` if the given string is a unary operator, ``False`` otherwise.
    """
    return string == '~'

def is_binary(string: str) -> bool:
    """Checks if the given string is a binary operator.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a binary operator, ``False`` otherwise.
    """
    return string == '&' or string == '|' or string == '->'
```

Notice that the definition of a propositional formula is very specific about the use of parentheses: '$(\phi\&\psi)$' is a valid formula, but '$\phi\&\psi$' is not and neither is '$((\phi\&\psi))$'; '$\sim\phi$' is a valid formula but '$(\sim\phi)$' is not, etc. These restrictive choices are made to ensure that there is a unique and easy way to **parse** a formula: to take a string that is a formula and figure out the complete **derivation tree** of how it is decomposed into simpler and simpler formulas according to the derivation rules from the above definition. Such a derivation tree is naturally expressed in a computer program as a tree data structure, and this book's pedagogical approach is to indeed implement it as such, so the bulk of the tasks of this chapter are focused on translating formulas back and forth between representation as a string and as an expression-tree data structure.

The file `propositions/syntax.py` defines a Python class `Formula` for holding a propositional formula as a data structure.

propositions/syntax.py

```
@frozen
class Formula:
    """An immutable propositional formula in tree representation, composed from
    atomic propositions, and operators applied to them.

    Attributes:
        root: the constant, atomic proposition, or operator at the root of the
            formula tree.
        first: the first operand to the root, if the root is a unary or binary
            operator.
        second: the second operand to the root, if the root is a binary
            operator.
    """
    root: str
    first: Optional[Formula]
    second: Optional[Formula]

    def __init__(self, root: str, first: Optional[Formula] = None,
                 second: Optional[Formula] = None):
        """Initializes a `Formula` from its root and root operands.

        Parameters:
            root: the root for the formula tree.
            first: the first operand to the root, if the root is a unary or
```

```
                binary operator.
            second: the second operand to the root, if the root is a binary
                operator.
        """
        if is_variable(root) or is_constant(root):
            assert first is None and second is None
            self.root = root
        elif is_unary(root):
            assert first is not None and second is None
            self.root, self.first = root, first
        else:
            assert is_binary(root)
            assert first is not None and second is not None
            self.root, self.first, self.second = root, first, second
```

The constructor of this class (which is already implemented) takes as arguments the components (between one and three) of which the formula is composed, and constructs the composite formula. For instance, to represent the formula '$(\phi\&\psi)$', the constructor will be given the three "components": the operator '&' that will serve as the "root" of the tree, and the two subformulas ϕ and ψ. Recall that the symbol '$\rightarrow$' will be represented in Python as the two-character sequence `'->'`.

Example. the data structure for representing '~(p&q76)' is constructed using the following code:

```
f = Formula('~', Formula('&', Formula('p'), Formula('q76')))
```

The different components of `f` from the above example can then be accessed using the instance variables `f.root` for the root, `f.first` for the first subformula (if any), and `f.second` for the second subformula (if any). To enable the safe reuse of existing formula objects as building blocks for other formula objects (and even as building blocks in more than one other formula object, or as building blocks that appear more than once in the same formula object), we have defined the `Formula` class to be **immutable**, i.e., we have defined it so that `f.root`, `f.first`, and `f.second` cannot be assigned to after `f` has been constructed. E.g., you can verify that after `f` is constructed as above, attempting to assign `f.first = Formula('q4')` fails. This is achieved by the `@frozen` decorator that appears just before the class definition.[2] Most of the classes that you will implement in this course will be made immutable in this way.

Your first task is to translate the expression-tree representation of a formula into its string representation. This is easily done using recursion: suppose that you know how to convert two tree data-structures `f` and `g` (that are both Python objects of type `Formula`) into strings; how can you convert, into such a string, a tree data structure of type `Formula` that has `'&'` at its root, and `f` and `g` as its two children/subformulas?

Task 1. Implement the missing code for the method `__repr__()` (of class `Formula`), which returns a string that represents the formula (in the syntax defined above). Note that in Python implementing this method also automatically defines that `str(f)` returns the same string as well.

[2]The definition of this decorator is in the file `logic_utils.py` that we have provided to you, and which we imported for you into `propositions/syntax.py`.

propositions/syntax.py
```
class Formula:
        ⋮
    def __repr__(self) -> str:
        """Computes the string representation of the current formula.

        Returns:
            The standard string representation of the current formula.
        """
        # Task 1.1
```

Example: For the formula `f` defined in the example above, `f.__repr__()` (and hence also `str(f)`) should return the string `'~(p&q76)'`.
Note: The decorator that precedes the definition of `__repr__()` in the code that you are given **memoizes** this method, so that any subsequent calls to this method (on the same `Formula` object) after the first call simply return the value returned by the first call instead of recalculating it. This has no effect on code correctness since the `Formula` class is immutable, but will dramatically speed-up your code in later chapters, when you handle complex formulas. We use this decorator throughout the code that you are given in various places where there are speed improvements to be gained.

The next two tasks ask for getting a summary of the components of a given formula: the variables used in it, and the operators used in it (where we treat 'T' and 'F' as operators too—we will discuss the rationale behind this definition in Chapter 3).

Task 2. Implement the missing code for the method `variables()` (of class `Formula`), which returns all of the atomic propositions that appear in the formula. Recall that an atomic proposition is a leaf of the tree whose label is a letter in 'p'…'z' optionally followed by a nonnegative integer.

propositions/syntax.py
```
class Formula:
        ⋮
    def variables(self) -> Set[str]:
        """Finds all atomic propositions (variables) in the current formula.

        Returns:
            A set of all atomic propositions used in the current formula.
        """
        # Task 1.2
```

Example: For the formula `f` defined in the example above, `f.variables()` should return `{'p', 'q76'}`.

Task 3. Implement the missing code for the method `operators()` (of class `Formula`), which returns all of the operators that appear in the formula. By operators we mean '~', '&', '|', '→', 'T', and 'F'.

propositions/syntax.py
```
class Formula:
        ⋮
    def operators(self) -> Set[str]:
        """Finds all operators in the current formula.
        Returns:
            A set of all operators (including 'T' and 'F') used in the current
            formula.
        """
        # Task 1.3
```

Example: For the formula `f` defined in the example above, `f.operators()` should return `{'~', '&'}`.

2 Parsing

Going in the opposite direction, i.e., taking a string representation of a formula and **parsing** it into the corresponding tree data structure is usually a bit more difficult, since you need to algorithmically figure out where to "break" the complex string representation of the formula into the different components of the formula. This type of parsing challenge is quite common when dealing with many cases of well specified "languages" that need to be "understood" by a computer program, the prime example being when compilers need to understand programs written in a programming language. There is a general theory that deals with various classes of languages as well as algorithms for parsing them, with an emphasis on the class of **context-free languages** whose grammar can be defined by a recursive definition. The language for valid formulas that we chose for this course is in this class and is simple enough so that a simple "recursive descent" algorithm, to be described below, will do the trick.

The idea is to first read the first **token** in the string, where a token is a basic "word" of our language: either one of the single-letter tokens `'T'`, `'F'`, `'('`, `')'`, `'~'`, `'&'`, `'|'`, or the two-letter "implies" token `'->'`, or a variable name like `'p'` or `'q76'`. This first token will tell you in a unique way how to continue reading the rest of the string, where this reading can be done recursively. For example, if the first token is an open parenthesis, `'('`, then we know that a formula ϕ must follow, which can be read by a recursive call. Once ϕ was recursively read, we know that the following token must be one of `'&'`, `'|'`, or `'->'`, and once this token is read then a formula ψ must follow, and then a closing parenthesis, `')'`. This should become concrete as you implement the following task:

Task 4. Implement the missing code for the function `_parse_prefix(string)` (a static method of class `Formula`), which takes a string that has a prefix that represents a formula, and returns a formula tree created from the prefix, and a string containing the unparsed remainder of the string (which may be empty, if the parsed prefix is in fact the entire string).

propositions/syntax.py

```
class Formula:
        ⋮
    @staticmethod
    def _parse_prefix(string: str) -> Tuple[Optional[Formula], str]:
        """Parses a prefix of the given string into a formula.

        Parameters:
            string: string to parse.

        Returns:
            A pair of the parsed formula and the unparsed suffix of the string.
            If the given string has as a prefix a variable name (e.g.,
            'x12') or a unary operator follows by a variable name, then the
            parsed prefix will include that entire variable name (and not just a
            part of it, such as 'x1'). If no prefix of the given string is a
            valid standard string representation of a formula then returned pair
            should be of ``None`` and an error message, where the error message
            is a string with some human-readable content.
        """
        # Task 1.4
```

Example: `Formula._parse_prefix('(p&q)')` should return a pair whose first element is a `Formula` object equivalent to `Formula('&', Formula('p'), Formula('q'))` and whose second element is `''` (the empty string), while `Formula._parse_prefix('p3&q')` should return a pair whose first element is a `Formula` object equivalent to `Formula('p3')` and whose second element is the string `'&q'`, and `Formula._parse_prefix('((p&q))')` should return the Python pair `(None, 'Unexpected symbol )')` (or some other error message in the second entry). See the test function `test_parse_prefix` in the file `propositions/syntax_test.py` for more examples (it is always a good idea to consult the test function for a task before starting to solve the task).

The fact that you were able to clearly decide, without any ambiguity, on what exactly is the prefix of the string that constitutes a valid formula relies on the fact that indeed our syntactic rules ensure that no prefix of a formula is also a formula itself (with the mentioned caveat that a variable name cannot be broken down so that only its prefix is taken, since, e.g., 'x1' *is* a prefix of 'x12'). Had our definitions been different, e.g. had we allowed '$\phi\&\psi$' as a formula as well, then this would have no longer been true. E.g., under such definitions, the string `'x&y'` would have been a valid formula, and so would have its prefix `'x'`. The code behind your implementation and the reasoning of why it solves the task in the unique correct way thus essentially proves the following lemma:

Lemma (Prefix-Free Property of Formulas)**.** *No formula is a prefix of another formula, except for the case of a variable name as a prefix of another variable name.*

Since this is the first lemma in the book, let us take just a moment to consider how this lemma would be proven in a "standard mathematical way." The overall structure of the proof would be by induction on the length of the formula (which we need to show has no proper prefix that is also a formula). The proof would then proceed with a case-by-case analysis of the first token of the formula. The significant parts of the proof would be the ones that correspond to the inductive definitions, specifically to a formula starting with a '('. By definition, this formula must be parsed as '$(\phi*\psi)$' (where $*$ is one of the three allowed binary operations), and so must any supposed formula prefix of it (for perhaps some other '$(\phi'*'\psi')$'). We would then use the induction hypothesis claiming that neither ϕ nor ϕ' can be the prefix of the other if they are different, to show that $\phi = \phi'$, which then forces $* = *'$, and then we can apply the induction hypothesis again to show that neither ψ nor ψ' can be the prefix of the other if they are different, to conclude the proof (of this case).[3] The structure of this proof is in direct correspondence

[3]The "caveat case" of a variable name as a prefix of another variable name would come up when dealing with formulas whose first token is a variable name (rather than with a '(' as in the case detailed above). In this case, to get uniqueness we must indeed enforce that the entire variable name token be part of the parsed prefix.

to your parsing algorithm and its justification: both have the same case-by-case analysis, only with mathematical induction in the proof replacing recursion in the algorithm. We thus feel that your solution of the task contains all the important mathematical content of the proof, missing only the formalistic wrapping, but having the advantage of being very concrete. In this book we will thus not provide formal mathematical proofs that just repeat in a formal mathematical way conceptual steps taken in a programmatic solution of a task.

Task 5. Implement the missing code for the function `is_formula(string)` (a static method of class `Formula`), which checks whether a given string is a valid formula (according to the definition above). Hint: use the `_parse_prefix()` method.

propositions/syntax.py

```
class Formula:
        ⋮
    @staticmethod
    def is_formula(string: str) -> bool:
        """Checks if the given string is a valid representation of a formula.

        Parameters:
            string: string to check.

        Returns:
            ``True`` if the given string is a valid standard string
            representation of a formula, ``False`` otherwise.
        """
        # Task 1.5
```

Task 6. Implement the missing code for the function `parse(string)` (a static method of class `Formula`), which parses a given string representation of a formula. (You may assume that the input string is valid, i.e., satisfies the precondition `Formula.is_formula(string)`, as indicated by the assertion that we already added for you.)

propositions/syntax.py

```
class Formula:
        ⋮
    @staticmethod
    def parse(string: str) -> Formula:
        """Parses the given valid string representation into a formula.

        Parameters:
            string: string to parse.

        Returns:
            A formula whose standard string representation is the given string.
        """
        assert Formula.is_formula(string)
        # Task 1.6
```

Hint: Use the `_parse_prefix()` method.

The reasoning and code that allowed you to implement Task 6 (and the preceding Task 4) without any ambiguity essentially proves the following theorem.

Theorem (Unique Readability of Formulas)**.** *There is a unique derivation tree for every valid propositional formula.*

3 Polish Notations

The notation that we used to represent our formulas is only one possible format, and there are other notations by which a tree data-structure can be represented as a string. The notation that we used is called **infix notation** since the operator at the root of the tree is given *in*-between the representations of the left and right subtrees. Another commonly used notation is **polish** notation.[4] In this notation, the operator is printed *before* the (two, in the case of a binary operator) subformulas that it operates on. Of course, these subformulas themselves are recursively printed in the same way. In another commonly used notation, **reverse polish notation**, the operator is printed *after* these subformulas.[5] One nice advantage of polish and reverse polish notations is that it turns out that parentheses are no longer needed. Thus, for example, the formula whose regular, infix, notation is '~(p&q76)' would be represented in polish notation as '~&pq76' and in reverse polish notation as 'pq76&~'.

Optional Task 7. Implement the missing code for the method `polish()` (of class `Formula`), which returns a string that represents the formula in polish notation.

propositions/syntax.py
```
class Formula:
        ⋮
    def polish(self) -> str:
        """Computes the polish notation representation of the current formula.

        Returns:
            The polish notation representation of the current formula.
        """
        # Optional Task 1.7
```

Example: For the formula `f` defined in the example above, `f.polish()` should return the string `'~&pq76'`. (Remember that there are no parentheses in polish notation.) Once again, it is always a good idea to consult the test function for more examples.

Parsing polish notation is usually a bit easier than parsing infix notation, even though there are no parentheses.

Optional Task 8. Implement the missing code for the function `parse_polish(string)` (a static method of class `Formula`), which parses a given polish notation representation of a formula. As in Task 6, you may assume (without checking) that the input string is valid.

propositions/syntax.py
```
class Formula:
        ⋮
    @staticmethod
    def parse_polish(string: str) -> Formula:
        """Parses the given polish notation representation into a formula.
```

[4]So called after the Polish logician Jan Łukasiewicz who invented it.

[5]Polish notation and reverse polish notations are also called **prefix notation** and **postfix notation**, respectively, analogously to infix notation, describing where the operator comes with respect to the representations of the subtrees. We avoid these terms here in order not to confuse prefix as the name of the notation with prefix as the word describing the beginning of a string as in "prefix-free" or as in `_parse_prefix()`.

```
        Parameters:
            string: string to parse.

        Returns:
            A formula whose polish notation representation is the given string.
        """
        # Optional Task 1.8
```

Hint: First implement an analogue of Task 4 for polish notation.

4 Infinite Sets of Formulas

Our programs, like all computer programs, only handle finite data. This book however aims to teach Mathematical Logic and thus need also consider infinite objects. We shall aim to make a clear distinction between objects that are mathematically finite (like a single integer number[6]) and those that can mathematically be infinite (like a set of integers) but practical representations in a computer program may limit them to be finite. So, looking at the definition of formulas, we see that every formula has a *finite* length and thus formulas are finite objects in principle. Now, there is no uniform bound on the possible length of a formula (much like there is no uniform bound on the possible length of an integer), which means that there are infinitely many formulas. In particular, a set of formulas can in principle be an infinite object: it may contain a finite or infinite number of distinct formulas, but each of these formulas has only a finite length. Of course, when we actually represent sets of formulas in our programs, the represented sets will always be only of finite size.

As the reader may recall, in Mathematics there can be different **cardinalities** of infinite sets, where the "smallest" infinite sets are called **countably infinite** (or **enumerable**). An infinite set S is called countable if there exists a way to list its items one after another without "forgetting" any of them: $S = \{s_1, s_2, s_3, \ldots\}$. (Formally if there exists a function f from the natural numbers *onto* S.)[7] The set of formulas is indeed countable in this sense: each formula is a finite-length string whose letters come from a finite number of characters, and thus there is a finite number of formulas of any given fixed length. Thus one may first list all the formulas of length 1, then those of length 2, etc. We thus get the following simple fact:

Theorem. *The set of formulas is countably infinite.*

While in our course there are only countably many variables and therefore only countably many formulas, all of our results extend naturally via analogous proofs to variable sets of arbitrary cardinality, which imply also formulas sets of arbitrary cardinality. In the few places throughout this book where the generalization is not straightforward, we will explicitly discuss this.

[6]Indeed, while there is no upper bound on the length of an integer number, any given single integer number is of finite length.

[7]For the benefit of readers who are not familiar with cardinalities of infinite sets, we note that while when first encountering this definition it may be hard to think of any set that does not satisfy this property, in fact many natural sets do not satisfy it. A prime example is the infinite set of all real numbers between 0 and 1, which is not countable.