4 Sentential Logic

THE FOCUS of this chapter is sentential logic . Sentential logic, also known as *propositional logic*, is concerned with zero-order sentences, such a sentence being either a boolean term, say, `(zero < S zero)`, or the result of applying one of the five sentential connectives (`not`, `and`, `or`, `if`, `iff`) to other zero-order sentences. Sentences that are boolean terms are viewed as indivisible units having no internal structure; the only meaning they are capable of having in sentential logic is a binary indication of truth or falsity, as will become clear when we come to discuss sentential semantics in Section 4.12.

Mastering sentential logic is a crucial step in becoming able to write and understand proofs in general, because sentential logic forms the core of more powerful systems such as first-order logic, which we will study in subsequent chapters, and indeed lies at the heart of mathematical reasoning in general. In this chapter we cover all available mechanisms for constructing proofs in sentential logic, giving numerous examples along the way; we develop a library of some useful sentential proof methods; we touch on some more advanced topics, such as recursive proof methods; and we introduce heuristics for constructing such proofs. We also give a computational treatment of the semantics of sentential logic, and we discuss some useful applications enabled by algorithms for solving the satisfiability (SAT) problem. Finally, we develop a theorem-proving method for sentential logic that will be used in the sequel. In all of the examples in this chapter we use the identifiers `A`, `B`, `C`, `D`, `E`, etc., as boolean atoms. We assume that appropriate declarations of these constants have already been made at the top level.

4.1 Working with the Boolean constants

The two Boolean atoms `true` and `false` are the ultimate foundations of sentential reasoning. We can derive `true` any time by applying the nullary method `true-intro`:

```
> (!true-intro)

Theorem: true
```

The constant `false` can only be derived if the assumption base is inconsistent. Specifically, if the assumption base contains two contradictory sentences of the form p and `(~ p)`, then applying the binary method `absurd` to p and `(~ p)` will derive `false`:

```
> assume A
    assume (~ A)
      (!absurd A (~ A))

Theorem: (if A
             (if (not A)
```

```
                false))
```

It is an error if the two arguments to `absurd` are not of the form p and `(~ p)`, respectively, or if either of them is not in the assumption base.

Finally, we can derive `(~ false)` at any time through the nullary method `false-elim`:

```
> (!false-elim)

Theorem: (not false)
```

4.2 Working with conjunctions

4.2.1 Using conjunctions: left-and and right-and

It is part and parcel of the meaning of a conjunction `(p & q)` that it logically implies each of its two components, p and q. If it is raining and it is windy, it follows that it is raining; and it also follows that it is windy. Accordingly, if we have a conjunction `(p & q)` in the assumption base, we should be able to derive p from it, as well as q. In Athena, the former is done by the unary method `left-and`. If the argument to `left-and` is a conjunction `(p & q)` that is present in the assumption base, then the output will be the conclusion p:

```
assert p := (A & B)

> (!left-and p)

Theorem: A
```

There is a similar unary method, `right-and`, that does the same thing for the right component of a conjunction: If `(p & q)` is in the assumption base,

```
(!right-and (p & q))
```

will generate the conclusion q. It is an error if the argument to `left-and` or `right-and` is not a conjunction, or if it is a conjunction that is not in the assumption base.

We say that `left-and` performs conjunction *elimination*,because it takes a conjunction as input and detaches its left component. Likewise for `right-and`. Most primitive deductive constructs of Athena either eliminate or else *introduce* one of the five logical connectives, or one of the quantifiers, thereby serving as introduction or elimination mechanisms for the corresponding connective or quantifier.

4.2.2 Deriving conjunctions: both

The introduction mechanism for conjunctions is the binary method `both`. Given any two sentences p and q in the assumption base, the method call

`(!both` p q`)`

will produce the conclusion `(`p `&` q`)`. As an example that uses all three methods dealing with conjunctions, consider the derivation of `(D & A)` from the premises `(A & B)` and `(C & D)`:

```
assert (A & B), (C & D)

> let {_ := conclude D
              (!right-and (C & D));
       _ := conclude A
              (!left-and (A & B))}
    (!both D A)

Theorem: (and D A)
```

Alternatively:

```
assert A-and-B := (A & B)
assert C-and-D := (C & D)

> {
   conclude D
     (!right-and C-and-D);
   conclude A
     (!left-and A-and-B);
   (!both D A)
  }

Theorem: (and D A)
```

or:

```
> (!both conclude D
           (!right-and C-and-D)
         conclude A
           (!left-and A-and-B))

Theorem: (and D A)
```

or even more tersely:

```
> (!both (!right-and C-and-D)
         (!left-and  A-and-B))

Theorem: (and D A)
```

4.3 Working with conditionals

4.3.1 Using conditionals: modus ponens and modus tollens

One of the most fundamental inference rules in logic is *modus ponens*, also known as *conditional elimination*. Starting from two premises of the form `(`p `==>` q`)` and p, modus ponens generates the conclusion q, thereby performing conditional elimination. In Athena, modus ponens is performed by the binary primitive method `mp`. When the first argument to `mp` is of the form `(`p `==>` q`)`, the second argument is p, and both arguments are in the assumption base, `mp` will successfully derive q. For instance, assuming that `(A ==> B)` and `A` are both in the assumption base, we have:

```
> (!mp (A ==> B) A)

Theorem: B
```

Another important method involving conditionals is *modus tollens*. Given two premises of the form `(`p `==>` q`)` and `(~` q`)`, modus tollens generates the conclusion `(~` p`)`. That's a valid form of reasoning. Suppose, for instance, that we know the following two sentences to be true:

1. If the keys are in the apartment, then they are in the kitchen.
2. The keys are not in the kitchen.

From these two pieces of information we can infer that the keys are not in the apartment.[1] That is the sort of inference licensed by modus tollens.

In Athena, modus tollens is performed by the binary method `mt`. When the first argument to `mt` is a conditional `(`p `==>` q`)`, the second is `(~` q`)`, and both are in the assumption base, `mt` will derive `(~` p`)`. For instance, assuming that `(A ==> B)` and `(~ B)` are both in the assumption base, we have:

1 Indeed, suppose that the keys are in the apartment. Then, by the first premise and modus ponens, they would have to be in the kitchen. But the second premise tells us that, in fact, they are not in the kitchen. We have thus arrived at two inconsistent conclusions: that the keys are in the kitchen and that they are not. This contradiction shows that our starting assumption is untenable—the keys cannot be in the apartment. This is an example of reasoning by contradiction, a topic that is discussed more extensively in Section 4.5.2.
An astute reader might point out that the derivation of a contradiction from our two premises along with the hypothesis that the keys are in the apartment does not necessarily compel us to retract that hypothesis. All it shows is that the three sentences together, as a whole, form an inconsistent set, and therefore that at least one of them must be false. Which one we choose to reject is up to us. That is true. We could restore consistency by rejecting one of the two premises instead. However, we generally operate under the assumption that we are not willing to give up the initial premises of an argument—the premises are taken for granted, and if any sentence p conflicts with them, then it is p that must be given up, not any of the premises. (Of course this is an otiose point if the starting premises are themselves inconsistent.)

```
> (!mt (A ==> B) (~ B))

Theorem: (not A)
```

4.3.2 Deriving conditionals: The `assume` construct

The standard way of proving a conditional (*p* `==>` *q*) is to assume the antecedent *p* (i.e., to add *p* to the current assumption base) and proceed to derive the consequent *q*. In Athena this is done with deductions of the form **`assume`** *p D*, where *D* is a proof that derives *q* from the augmented assumption base. To illustrate this technique, here is a proof of `(A ==> A)`:

```
> assume A
    (!claim A)

Theorem: (if A A)
```

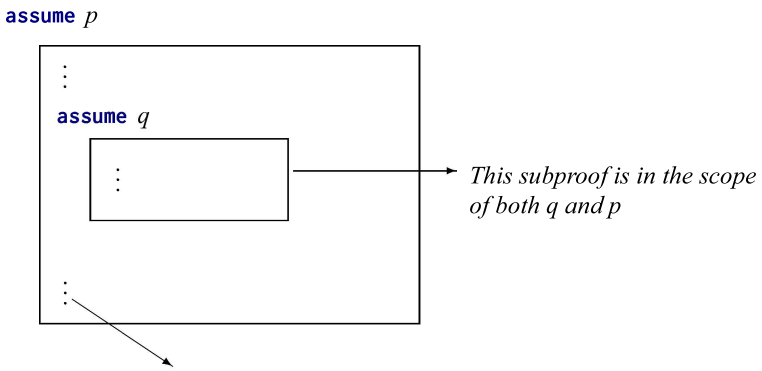
Deductions of the form **`assume`** *p D* are called *conditional* or *hypothetical* deductions. We refer to *p* and *D* as the *hypothesis* (or *assumption*) and the *body* of the conditional deduction, respectively. We also say that *D* represents the *scope* of the hypothesis *p*. So, in the above example, the hypothesis is the atom `A`, and its scope is the deduction `(!claim A)`, which constitutes the body of the conditional proof.

To evaluate a deduction of the form

`assume` *p D* (4.1)

in an assumption base β, we add *p* to β and go on to evaluate the body *D* in the augmented assumption base $\beta \cup \{p\}$. The fact that *D* is evaluated in $\beta \cup \{p\}$ means that *the assumption p can be freely used anywhere within its scope*, that is, anywhere inside *D*. If and when the evaluation of *D* in $\beta \cup \{p\}$ produces a conclusion *q*, we return the conditional (*p* `==>` *q*) as the result of (4.1). If the evaluation of *D* in $\beta \cup \{p\}$ diverges or results in an error, then the evaluation of (4.1) also diverges or results in that same error, respectively.

The body *D* is said to be a *subproof* (or *subdeduction*) of the conditional proof (4.1). Subproofs can be nested inside one another, as the figure below illustrates. Note that, strictly speaking, the scope of a hypothesis is not determined by indentation but rather by the syntax rules that define what counts as a deduction. Nevertheless, proper indentation can be very helpful in demarcating the scope of an assumption (and, more generally, in clarifying the structure of a proof). Typically, the body *D* of a hypothetical deduction **`assume`** *p D* is written on the line after the hypothesis *p* and indented two spaces to the right of the keyword **`assume`**.



For instance, in the following proof the body `(!claim A)` is in the scope of both the inner assumption `B` and the outer assumption `A`:

```
> assume A
    assume B
      (!claim A)

Theorem: (if A
             (if B A))
```

Note that the contents of the assumption base are immaterial for the above proof. Evaluating this proof in any β whatsoever will successfully produce the result

```
(A ==> B ==> A).
```

That is the case for all and only those sentences that are *tautologies*. A tautology is precisely a sentence that can be derived from every assumption base. Given that Athena's classical logic is monotonic,[2] an alternative way of saying the same thing is this: A tautology is a sentence that can be derived from the empty assumption base. The equivalence of the two conditions can be shown as follows. In one direction, if p can be derived from every assumption base, then it can certainly be derived from the empty assumption base. Conversely, if p can be derived from the empty assumption base then, by monotonicity, it can be derived from every assumption base.[3]

2 Monotonicity means that if p is derivable from some β, then it is also derivable from every superset $\beta' \supseteq \beta$. Intuitively, the import of this is that *adding* new information to an assumption base does not invalidate any conclusions previously obtainable from it.

3 This is a syntactic or proof-theoretic characterization of tautologies. A semantic characterization would say that a tautology is a sentence that is true under all interpretations, or, alternatively, logically entailed by the empty assumption base; we will pursue the semantic direction later on, in Section 4.12. Since the proof system of Athena

Oftentimes it is useful to give a name to the hypothesis of a conditional deduction, and then use that name inside the body to refer to the hypothesis. This can be done with the following variant of the **assume** construct:

assume $I := p$
  D

which is really a shorthand for:

let $\{ I := p \}$
  assume I
    D

As an example, consider the proof of the tautology `(A & B ==> B & A)`:

```
> assume hyp := (A & B)
    let {left  := (!left-and hyp);
         right := (!right-and hyp)}
      (!both right left)

Theorem: (if (and A B)
             (and B A))
```

Actually, Athena allows a simpler proof:

```
> assume (A & B)
    (!both B A)

Theorem: (if (and A B)
             (and B A))
```

This proof is possible because whenever the hypothesis of a conditional proof is a conjunction, Athena not only adds that conjunction to the assumption base, but it also adds each of the conjuncts (and the conjuncts of those conjuncts, etc.). Thus, in the above case, both `A` and `B` are in the assumption base when `both` is invoked. A similar automatic decomposition of conjunctions also happens during intermediate proof steps in a **let** deduction. For instance:

```
assert A=>B&C := (A ==> B & C)
assert A

let {B&C := (!mp A=>B&C A);
     ... # We now have not just (B & C), but also B and C in the a.b.
    }
  ...
```

is complete (meaning that a sentence p is entailed by an assumption base β iff it is derivable from β), these two views of tautologies coincide.

Consider next this implication:

```
((A ==> B ==> C) ==> (B ==> A ==> C)).
```
(4.2)

The following proof derives (4.2):

```
1  > assume hyp := (A ==> B ==> C)
2      assume B
3        assume A
4          let {B=>C := (!mp hyp A)}
5            conclude C
6              (!mp B=>C B)
7
8  Theorem: (if (if A
9                   (if B C))
10               (if B
11                  (if A C)))
```

Observe that the structure of the proof closely mirrors the structure of the goal. The proof is constructed by reasoning as follows: Our top goal, (4.2), is a conditional of the form `(`p `==>` q`)`, hence our proof, starting on the first line, will be of the form **`assume`** p D, where D must be a proof capable of deriving the subgoal q. Now, in this case q is also a conditional, namely, `(B ==> A ==> C)`, so D, starting on line 2, will be a deduction of the form

`assume` `B` D',

where D' is a deduction that will derive the subgoal `(A ==> C)`. Since that subgoal is again a conditional, the proof D' that starts on line 3 is of the form

`assume` `A` D'',

where D'' now has to derive the subgoal `C` from the preceding three assumptions. This can be done by the two inferences on lines 4 and 6, first by deriving `(B ==> C)` and finally by deducing `C`; both steps use modus ponens on the appropriate premises. As we discuss in Section 4.14, most proofs are constructed through similar combinations of reasoning that is driven exclusively by the syntactic form of the goal and reasoning that is driven by the contents of the assumption base.

Of course, a conditional may be derived in many other ways. For instance, if the assumption base contains a conjunction of the form `(`p `&` `(`q `==>` r`))`, then it is simpler and more direct to derive the conditional `(`q `==>` r`)` by performing right conjunction elimination on that premise instead of using **`assume`**. The same goes for sentences of other forms, say, for biconditionals. The proof techniques discussed in this chapter are the canonical or most common ways of inferring the corresponding kinds of sentences, but, depending on the context, it may be simpler to use an alternative method.

4.4 Working with disjunctions

4.4.1 Using disjunctions: Reasoning by cases

Suppose we are trying to derive some goal p. If the assumption base contains a disjunction $(p_1 \mid p_2)$, we can often put that disjunction to use as follows: We know that p_1 holds or p_2 holds. If we can show that p_1 implies the goal p *and* that p_2 also implies p, then we can conclude p. For, if p_1 holds, then p follows from the implication $(p_1 \texttt{ ==> } p)$, p_1, and modus ponens; while, if p_2 holds, then p follows from $(p_2 \texttt{ ==> } p)$, p_2, and modus ponens. This type of reasoning (called "case analysis," or "reasoning by cases") is pervasive, both in mathematics and in real life.[4]

In Athena this type of reasoning is carried out by the ternary method `cases`. The first argument of this method must be a disjunction, say $(p_1 \mid p_2)$; while the second and third arguments must be conditionals of the form $(p_1 \texttt{ ==> } p)$ and $(p_2 \texttt{ ==> } p)$, respectively. If all three sentences are in the assumption base, then the conclusion p is produced as the result. It is an error if the arguments are not of the form just described, or if one of them is not in the assumption base. For instance:

```
assert (C1 | C2), (C1 ==> B), (C2 ==> B)

> conclude B
    (!cases (C1 | C2)
            (C1 ==> B)
            (C2 ==> B))

Theorem: B
```

As another example, suppose the assumption base contains `(C ==> B)` and `((A & B) | C)` and from these two premises we want to derive the conclusion `B`. We can do that with the following case analysis:

```
> (!cases (A & B | C)
    assume (A & B)
      (!claim B)
    assume C
      (!mp (C ==> B) C))

Theorem: B
```

4 For instance, suppose we know that either the transmission of a car is broken or the battery needs to be replaced. And we also know that the cost of fixing the transmission exceeds $150, while replacing the battery also costs more than $150. Then we can conclude that in either case, fixing the car will cost more than $150.

Note how the implications required by cases in its second and third argument positions are naturally produced as results of **assume** deductions. And in the first case, `(!claim B)` succeeds because, as noted earlier, **assume** `(A & B)` adds the conjuncts `A` and `B` to the assumption base, along with `(A & B)`.

A useful variant of reasoning by cases performs a case analysis on a sentence p and its negation `(~ p)`. We know that for any given p, either p or `(~ p)` holds; this is the law of the *excluded middle*. Therefore, if we can show that a goal q follows both from p and from `(~ p)`, we should be able to conclude q. This is done with the binary method `two-cases`, which takes two premises of the form `(p ==> q)` and `(~ p ==> q)` and derives q. For example:

```
assert (A ==> B), (~ A ==> B)

> (!two-cases
    (A ==> B)
    (~ A ==> B))

Theorem: B
```

The two premises can be passed in either order, for example, the negative conditional can be the first argument and the positive second:

```
> (!two-cases
    (~ A ==> B)
    (A ==> B))

Theorem: B
```

This is not a primitive Athena method. It is defined in terms of cases as follows:

```
define two-cases :=
  method (cond-1 cond-2)
    let {M := method (p q)
                (!cases conclude (p | ~ p)
                         (!ex-middle p)
                  (p ==> q)
                  (~ p ==> q))}
      match [cond-1 cond-2] {
        [(p ==> q) ((~ p) ==> q)] => (!M p q)
      | [((~ p) ==> q) (p ==> q)] => (!M p q)
    }
```

where `ex-middle` is a unary method that takes an arbitrary sentence p and derives the theorem `(p | ~ p)`. (We ask you to implement this method in Exercise 4.15.)

4.4.2 Deriving disjunctions

The most straightforward way to derive a disjunction `(`p `|` q`)` is to derive the left component, p, or the right component q. If we have p in the assumption base, then `(`p `|` q`)` can be derived by applying the binary method `left-either` to p and q:

`(!left-either` p q`)`.

Likewise, if q is the assumption base, then the method call

`(!right-either` p q`)`

will derive the disjunction `(`p `|` q`)`. For instance:

```
> assume A
    (!claim A)

Theorem: (if A A)

> (!left-either (A ==> A) B)

Theorem: (or (if A A)
             B)

> (!right-either B (A ==> A))

Theorem: (or B
             (if A A))
```

In addition to these two primitive methods, Athena offers a third, more versatile mechanism for disjunction introduction, the binary method `either`. If either p *or* q is in the assumption base, then `(!either` p q`)` derives the disjunction `(`p `|` q`)`. Otherwise, if neither argument is in the assumption base, `either` fails.

In realistic proofs, one is unlikely to derive a disjunction simply by deriving one of the two disjuncts. Usually there is a relationship between the two that must be taken into account, and that relationship is lost when each disjunct is treated in isolation from the other. Consider, for instance, the derivation of `(A | ~ A)` from the empty assumption base. Neither `A` nor `(~ A)` is derivable by itself, but their disjunction is.

There are a few different avenues for dealing with such cases. One of them is to reason by contradiction, that is, to negate the entire disjunction and proceed to derive a contradiction. In the foregoing example, that would mean assuming `(~ (A | ~ A))` and trying to derive a contradiction from that. (Reasoning by contradiction is discussed in Section 4.5.2.) Another useful technique is to treat a disjunction `(`p `|` q`)` as the conditional

`(~` p `==>` q`)`

and use the corresponding introduction mechanism for conditionals (Section 4.3). The conditional (~ p ==> q) is logically equivalent to the disjunction (p | q), so if we succeed in deriving the former, we should be able to derive the latter as a matter of routine. Indeed, in Section 4.9 we develop a method that can take an arbitrary conditional of the form (~ p ==> q) and produce the corresponding disjunction (method `cond-def`, page 224). With that method under our belt, we can transform a disjunction-derivation problem into a conditional-derivation problem.

4.5 Working with negations

4.5.1 Using negations

The only primitive method for negation elimination is `dn`, which stands for "double negation." It is a unary method, whose argument must be of the form (~ ~ p). If that sentence is in the assumption base, then the call

(!dn (~ ~ p))

will produce the conclusion p, thereby eliminating the two leading negation signs. For instance:

```
> assume h := (~ ~ A)
    (!dn h)

Theorem: (if (not (not A))
             A)
```

There are two other primitive methods that require some of its arguments to be negations: `mt` and `absurd`. Therefore, depending on what exactly we are trying to prove, we might be able to use a negation in the assumption base as an argument to one of these methods.

4.5.2 Deriving negations: Proof by contradiction

Proof by contradiction[5] is one of the most useful and common forms of deductive reasoning. The basic idea is to establish a negation (~ p) by assuming p and showing that this assumption (perhaps in tandem with other working assumptions) leads to an absurdity, namely, to `false`. That entitles us to reject the hypothesis p and conclude the desired (~ p).

The binary method `by-contradiction` is one way to perform this type of reasoning in Athena. The first argument to `by-contradiction` is simply the sentence we are trying to establish, typically a negation (~ p). The second argument must be the conditional

5 Also known as *reductio ad absurdum* and *indirect proof*.

(*p* `==>` `false`), essentially stating that the hypothesis *p* leads to an absurdity. If that conditional is in the assumption base, then the desired conclusion (`~` *p*) will be produced.

As an example, suppose that the assumption base contains the premises (`A ==> B & C`) and (`~ B`), and from these two pieces of information we want to derive the negation (`~ A`). We can reason by contradiction as follows: Suppose `A` holds. Then, by the first premise and modus ponens, we would have (`B & C`), and hence, by conjunction elimination, `B`. But this contradicts the second premise, (`~ B`), and that contradiction allows us to reject the hypothesis `A`, concluding (`~ A`). In Athena code:

```
assert premise-1 := (A ==> B & C)
assert premise-2 := (~ B)

> (!by-contradiction (~ A)
    assume A
      let {p1 := conclude (B & C)
                  (!mp premise-1 A);
           _ := conclude B
                  (!left-and p1)}
        (!absurd B premise-2))

Theorem: (not A)
```

Of course, we could have derived the conditional (`A ==> false`) first, and then, in a separate step, deduced (`~ A`) by applying `by-contradiction`:

```
> assume A
    let {_ := conclude (B & C)
                (!mp premise-1 A)}
      (!absurd B premise-2)

Theorem: (if A false)

> (!by-contradiction (~ A)
     (A ==> false))

Theorem: (not A)
```

However, we believe that the first version is more readable, so typically the requisite conditional (*p* `==>` `false`) will be established by a hypothetical deduction of the form

assume *p D*

that will appear directly as the second argument to `by-contradiction`. Observe that the semantics of nested deductions that we discussed in Section 2.10.2 are crucial here. A proof like

```
(!by-contradiction (~ p)
   assume p ···)
```

works precisely because the **assume**, being a deductive argument to the outer method call, will have its conditional conclusion incorporated into the assumption base by the time by-contradiction is applied.

Recall that the most direct way to derive false is to apply the binary method absurd to two contradictory sentences of the form q and (~ q) in the assumption base. For instance, assuming that A and (~ A) are both in the assumption base, we have:

```
> (!absurd A (~ A))

Theorem: false
```

Accordingly, a proof of (~ p) by contradiction often has the following logical structure:

```
(!by-contradiction (~ p)
   assume p
     let {p1 := conclude q
                  D1;
          p2 := conclude (~ q)
                  D2}
       (!absurd p1 p2))
```

In many cases, however, one of the two contradictory sentences, q or (~ q), is already in the assumption base, so we do not have to deduce it explicitly. We simply derive the other element of the contradictory pair and then move directly to the absurd application.

As another example, here is a proof that derives (~ B) from (~ (A ==> B)):

```
assert premise := (~ (A ==> B))

> (!by-contradiction (~ B)
    assume B
      let {A==>B := assume A
                      (!claim B)}
        (!absurd A==>B premise))

Theorem: (not B)
```

Sometimes the sentence p that we want to establish by contradiction is not a negation; it might be an atom, or a disjunction, or something else. That would seem to present a problem, since, according to what we have said so far, by-contradiction can only introduce negations. However, in classical logic every sentence p is equivalent to the double negation (~ ~ p), so this is not really a limitation: We can simply infer (~ ~ p) by assuming (~ p) and deriving a contradiction. After that, we can eliminate the double negation sign with

dn. For example, suppose that the assumption base contains A and (~ (A & ~ B)), and we want to derive the atom B. Following this recipe, we have:

```
assert premise-1 := (~ (A & ~ B))
assert premise-2 := A

> let {--B := (!by-contradiction (~ ~ B)
                  assume (~ B)
                   (!absurd (!both A (~ B)) premise-1))}
   (!dn --B)

Theorem: B
```

However, this routine of inferring a double negation first and then eliminating the two leading negation signs is somewhat tedious. For that reason, by-contradiction offers a shortcut: When the conclusion p to be established is not in the explicit form of a negation, it suffices to establish the conditional (~ p ==> false). We can then apply by-contradiction directly to p and this conditional:

```
(!by-contradiction p
   (~p ==> false))
```

and the desired p will be obtained. Thus, for instance, the foregoing proof could be more concisely written as follows:

```
> (!by-contradiction B
     assume (~ B)
       (!absurd (!both A (~ B)) premise-1))

Theorem: B
```

Let us say that the *complement* of a negation (~ p) is the sentence p, while the complement of a sentence p that is not a negation is the sentence (~ p). Thus, for example, the complement of A is (~ A), while the complement of (~ A) is A. We can define complementation in Athena with the following procedure:

```
define (complement p) :=
   match p {
     (~ q) => q
    | _ => (~ p)
   }
```

We write $\overline{p}$ to denote the complement of a sentence p. Note that $\overline{\overline{p}} = p$.

With this notion, we can succinctly specify the behavior of by-contradiction as follows. If the conditional ($\overline{p}$ ==> false) is in the assumption base, then

(!by-contradiction p

(!$\overline{p}$ ==> false))

will produce the conclusion p; otherwise it will fail.

There are two other auxiliary methods for reasoning by contradiction:

1. The unary method `from-false` derives any given sentence, provided that the assumption base contains `false`. That is, (`!from-false` p) will produce the theorem p whenever the assumption base contains `false`. This captures the principle that "everything follows from `false`."
2. The ternary method `from-complements` derives any given sentence p provided that the assumption base contains two complementary sentences q and $\overline{q}$. Specifically,

   (`!from-complements` p q $\overline{q}$)

   will derive p provided that both q and $\overline{q}$ are in the assumption base. Such an application can be read as: "Infer p from the complements q and $\overline{q}$."

These two methods are not primitive. Both are defined in terms of `by-contradiction`, and indeed each is definable in terms of the other; see Section 4.9 and Exercise 4.10. Nevertheless, it is convenient to have both available as separate methods.

4.6 Working with biconditionals

4.6.1 Using biconditionals

There are two elimination methods for biconditionals, `left-iff` and `right-iff`. For any given biconditional (p <==> q) in the assumption base, the method call

(`!left-iff` (p <==> q))

will produce the conclusion (p ==> q), while

(`!right-iff` (p <==> q))

will yield (q ==> p). For instance:

```
assert bc := (A <==> B)

> (!left-iff bc)

Theorem: (if A B)

> (!right-iff bc)

Theorem: (if B A)
```

4.6.2 Deriving biconditionals

The introduction method for biconditionals is `equiv`. Given two conditionals `(`p `==>` q`)` and `(`q `==>` p`)` in the assumption base, the call

`(!equiv (`p `==>` q`) (`q `==>` p`))`

will derive the biconditional `(`p `<==>` q`)`:

```
assert (A ==> B), (B ==> A)

> (!equiv (A ==> B) (B ==> A))

Theorem: (iff A B)
```

Note that there is a similarity between the introduction and elimination methods for conjunctions and those for biconditionals, with `left-and` and `right-and` being analogous to `left-iff` and `right-iff`, respectively, and with `both` analogous to `equiv`. This similarity is no accident, as a biconditional `(`p `<==>` q`)` is logically equivalent to the conjunction of the two conditionals `(`p `==>` q`)` and `(`q `==>` p`)`. Indeed, if we regard `(`p `<==>` q`)` as syntax sugar for the conjunction `((`p `==>` q`) & (`q `==>` p`))`, the analogy becomes exact.

4.7 Forcing a proof

When we set out to write a proof, we often know the rough outline of the proof's structure, but we don't know ahead of time how to fill in the details. (This is a recurrent theme that will come to the forefront when we turn to proof heuristics in Section 4.14.) Suppose, for example, that our goal is some sentence p, and that we have decided to proceed by a case analysis on a disjunction `(`p_1 `|` p_2`)`. Accordingly, we know that the general form of the proof will be

```
(!cases (p1 | p2)
  assume p1
    conclude p
      D1
  assume p2
    conclude p
      D2)
```

although we have not yet figured out what D_1 and D_2 should be. The fact that we don't yet have a complete proof should not prevent us from entering into the system the partial outline that we do have and verifying that it properly derives the goal. Later on we will make additional progress on D_1 and on D_2, and we will want to evaluate that progress as well. In general, proof headway is made incrementally, and we should have a way of

testing these contributions as they are being made, gradually, instead of having to wait until we have a finished proof with all the details filled in. Yet Athena can only evaluate syntactically complete deductions. So it would seem, for example, that the only way to evaluate the preceding proof outline will be to expand D_1 and D_2 into complete deductions. How can we resolve that tension?

The answer lies in the built-in primitive method **force**. This is a quite special unary method (hence the distinct coloring) that takes an arbitrary sentence p as its argument and produces p right back as its result. It is, in a sense, the deductive equivalent of the identity function, and it is therefore similar to the reiteration method `claim`. Unlike `claim`, however, the **force** method does not bother to check that the given sentence is in the assumption base. In fact, **force** doesn't perform any checking at all; it simply returns whatever sentence p it is given as a theorem. It should be clear that this is an unsound and generally dangerous method that should not be used anywhere in a *finished*, complete proof. However, it can be a useful tool during proof development, for the reasons described. For example, using **force** we can express the foregoing proof sketch as follows:

```
(!cases (p1 | p2)
  assume p1
    conclude p
      (!force p)
  assume p2
    conclude p
      (!force p))
```

This a proof that can be evaluated perfectly well on its own and will produce the intended conclusion p—as long as the `cases` application is correct. Once we have tested this outer proof skeleton, we can go about replacing the first **force** application with another proof sketch, and so on, until eventually there are no occurrences of **force** left anywhere in the proof text, and we have a complete proof that successfully derives p.[6] Using methods we could decompose and subdivide the work further in a more structured way, for example:

```
(!cases (p1 | p2)
  assume p1
    conclude p
```

6 Another alternative would be to insert dummy deductions in place of D_1 and D_2, provided we remove the "**conclude** p" annotations, for example, we might replace both D_1 and D_2 by `(!true-intro)`. This would enable us to evaluate the overall deduction. Then, as we begin to fill in the details, these bogus `(!true-intro)` applications would be removed. However, that approach has two serious drawbacks, apart from requiring the deletion of the **conclude** annotations. First, the result of the entire proof would become `true`, rather than the intended p. This means that we cannot have this partial proof embedded in a larger context (a surrounding partial proof) in which it would be expected of it to produce the proper conclusion, p. Second, we cannot patch up the proof in a truly incremental fashion. For instance, we would be forced to remove both `true-intro` applications simultaneously, because if we remove only the first one, replacing it by a proper deduction D_1 that derives p, while keeping the second `true-intro` application, the `cases` application would fail. The use of **force** averts both of these complications.

```
      (!handle-case-1 p1 p)
  assume p2
    conclude p
      (!handle-case-2 p2 p))
```

where the case handlers could be defined as independent methods, initially defined simply as applications of **force**:

```
define handle-case-1 :=
  method (premise goal)
   # Assume the premise holds, derive the goal
    (!force goal)

⋮

define handle-case-2 :=
  method (premise goal)
    (!force goal)
```

and the removal of the **force** applications could now proceed incrementally in these separate methods.

4.8 Putting it all together

In this section we will work through a proof that illustrates several of the mechanisms discussed in this chapter. Suppose we are given the following two premises:

```
assert premise-1 := (A & B | (A ==> C))

assert premise-2 := (C <==> ~ E)
```

and our task is to write a proof D that derives the conditional `(~ B ==> A ==> ~ E)` from the two premises.

Since our goal is a conditional, we will derive it with a conditional proof that assumes the antecedent and derives the consequent, so our top-level deduction D will be of the form:

```
assume (~ B)
  D'
```

where D' must derive the subgoal `(A ==> ~ E)` from the hypothesis `(~ B)` and the two premises. Now this subgoal is itself a conditional, so D' will be a conditional proof as well. Thus, the skeleton of our top-level proof D now takes the following form:

```
assume (~ B)
  assume A
```

```
  D''
```

where D'' needs to derive the subgoal `(~ E)` from the two previous hypotheses, `A` and `(~ B)`, along with the given premises.

We can now proceed in two ways. Continuing our "backward" or "goal-driven" analysis (more on that in Section 4.14), we could look at the form of the current subgoal and use the corresponding introduction mechanism for it. The current subgoal is a negation, `(~ E)`, so we would use proof by contradiction to derive it, and D'' would take the following form:

```
(!by-contradiction (~ E)
  assume E
    D''')
```

where D''' would have to derive `false`.

The second way to proceed is to examine the information that is available to D'' and see whether we can derive the desired conclusion `(~ E)` in a more direct manner. Let us pursue this second alternative here. D'' has access to four sentences: the two starting premises and the two hypotheses `(~ B)` and `A`. How can we use these to derive `(~ E)`? Well, the first premise is a disjunction,

```
(A & B | (A ==> C)),
```

and we saw that the chief way to use a disjunction `(p | q)` in order to infer a goal r is to perform a case analysis: Show that the goal r is implied both by p and by q. So in this case we would have to show that `(~ E)` follows both from the first disjunct `(A & B)` and from the second one, `(A ==> C)`. Accordingly, D'' takes the following form:

```
(!cases premise-1
  assume (A & B)
    conclude (~ E)
      D1
  assume (A ==> C)
    conclude (~ E)
      D2)
```

where now D_1 has to derive `(~ E)` from the four available sentences along with `(A & B)`; while D_2 has to derive `(~ E)` from the four sentences along with `(A ==> C)`. These deductions are now fairly easy to construct. Let us start with D_1. The hypothesis `(A & B)` gives us `B`, since `assume (A & B)` places `A` and `B` in the assumption base. But we are currently operating within the scope of the hypothesis `(~ B)`, so this contradiction allows us to conclude the desired `(~ E)` by a straightforward use of `from-complements`. D_1 thus becomes:

```
(!from-complements (~ E) B (~ B))
```

Continuing with D_2: By the second premise, `C` is equivalent to `(~ E)`, so we can detach the conditional `(C ==> ~ E)` via `left-iff`. But the innermost hypothesis `(A ==> C)` along with the preceding hypothesis `A` gives us `C`, so `(~ E)` can now be obtained simply by `mp`:

```
let {C1 := conclude (C ==> ~ E);
             (!left-iff premise-2);
     C2 := conclude C
             (!mp (A ==> C) A)}
conclude (~ E)
  (!mp C1 C2)
```

Assembling all the pieces together, and using a little additional naming, we arrive at the following final version:

```
assert premise-1 := (A & B | (A ==> C))
assert premise-2 := (C <==> ~ E)

assume -B := (~ B)
  assume A
    conclude -E := (~ E)
      (!cases premise-1
              assume (A & B)
                (!from-complements -E B -B)
              assume A=>C := (A ==> C)
                let {C=>-E := (!left-iff premise-2);
                     C     := (!mp A=>C A)}
                  (!mp C=>-E C))
```

In Section 4.14 we present heuristics for performing this kind of proof-finding analysis more systematically.

4.9 A library of useful methods for sentential reasoning

The inference mechanisms we have covered so far constitute a complete deduction system for sentential logic, meaning that if any zero-order sentence p follows logically from a finite set of zero-order sentences β, then there exists a proof expressed in terms of these mechanisms that derives p from β. However, these mechanisms are too low-level, in that the inference steps we can take with them are very small. Each such mechanism either eliminates or else introduces a logical connective. But expressing a realistic proof exclusively in terms of elimination and introduction steps can be unduly cumbersome. Very often we would like to take a larger inferential step in one fell swoop. To take but one example, we might want to apply one of De Morgan's laws to directly infer

$$(\sim p \ \& \sim q) \tag{4.3}$$