would represent the identity function.[18]

2.10 Expressions and deductions

In this section we discuss some common kinds of expressions and deductions. The most basic kind of expression is a *procedure call*, or procedure application. The general syntax form of a procedure call is:

$$(E\ F_1 \cdots F_n) \tag{2.6}$$

for $n \geq 0$, where E is an expression whose value must be a procedure and the arguments $F_1 \cdots F_n$ are phrases whose values become the inputs to that procedure.

Similarly, the most basic kind of deduction is a *method call*, or method application. The syntax form of a method call is

$$(\texttt{apply-method}\ E\ F_1 \cdots F_n) \tag{2.7}$$

where E is an expression that must denote a *method* M and the arguments $F_1 \cdots F_n$, $n \geq 0$, are phrases whose values will become the inputs to M. The exclamation mark `!` is typically used as a shorthand for the reserved word **`apply-method`**, so the most common syntax form of a method call is this:

$$(!E\ F_1 \cdots F_n). \tag{2.8}$$

Observe that both in (2.6) and in (2.7) the arguments are *phrases* $F_1, \ldots, F_n$, which means that they can be *either* deductions *or* expressions.[19] By contrast, that which gets applied in both cases must be the value of an *expression* E, the syntactic item immediately preceding the arguments $F_1 \cdots F_n$. The reason why we have an expression E there and not a phrase F is that deductions can only produce sentences, and what we need in that position is something that can be applied to arguments—either a procedure or a method, not a sentence.

To evaluate a method call of the form (2.7) in an assumption base β, we first need to evaluate E in β and obtain some method M from it; then we evaluate each argument phrase F_i in β, obtaining some value V_i from it (with a small but important twist that we discuss in Section 2.10.2); and finally we apply the method M to the argument values $V_1, \ldots, V_n$. What exactly is a method? A method can be viewed as the deductive analogue of a procedure: It takes a number of input values, carries out an arbitrarily long *proof* involving those values, and eventually yields a certain *conclusion* p. Alternatively, the method might (a) halt with an error message, or (b) diverge (i.e., get into an infinite loop). These are the only three possibilities for the outcome of a method application.

18 Note that `Lambda` is different from the keyword **`lambda`**. Athena identifiers are case-sensitive.

19 Recall that a phrase F is either an expression E or a deduction D.

The simplest available method is the nullary method `true-intro`. Its result is always the constant `true`, no matter what the assumption base is:

```
> (!true-intro)

Theorem: true
```

The next simplest method is the unary reiteration method `claim`. This method takes an arbitrary sentence *p* as input, and if *p* is in the assumption base, then it simply returns it back as the output:

```
1 assert true
2
3 > (!claim true)
4
5 Theorem: true
```

Note that the result of any deduction *D* is always reported as a *theorem*. That is because the result of *D* is guaranteed to be a logical consequence of the assumption base in which *D* was evaluated.

Every theorem produced by evaluating a deduction at the top level is automatically added to the global assumption base. In this case, of course, the global assumption base already contains the sentence `true` (because we asserted it on line 1), so adding the theorem `true` to it will not change its contents.

Claiming a sentence *p* will succeed if and *only* if *p* itself is in the assumption base. Suppose, for instance, that starting from an empty assumption base we **`assert`** *p* and then we claim $(p \mid p)$. We cannot expect that claim to succeed, despite the fact that $(p \mid p)$ is logically equivalent to *p*. The point here is that `claim` will check to see if *its input sentence p* is in the assumption base, not whether the assumption base contains some sentences from which *p* could be inferred, or some sentence that is equivalent to *p*. For example:

```
clear-assumption-base

assert true

> (!claim true)

Theorem: true

> (!claim (true | true))

Error, standard input, 1.9: Failed application of claim---the sentence
(or true true) is not in the assumption base.
```

More precisely, the identity required by `claim` is alpha-equivalence. That is, applying `claim` to *p* will succeed iff the assumption base contains a sentence that is alpha-equivalent to *p*:

```
assert (forall ?x . ?x = ?x)

> (!claim (forall ?y . ?y = ?y))

Theorem: (forall ?y:'S
           (= ?y:'S ?y:'S))
```

Alpha-equivalence is the relevant notion of identity for all primitive Athena methods.

Most other primitive Athena methods are either *introduction* or *elimination* methods for the various logical connectives and quantifiers. We describe their names and semantics in detail in chapters 4 and 5, and also in Appendix A, but we will give a brief preview here, starting with the methods for conjunction introduction and elimination. Conjunction introduction is performed by the binary method `both`, which takes any two sentences p and q, and provided that both of them are in the assumption base (again, up to alpha-equivalence), it produces the conclusion (and p q):

```
declare A, B, C: Boolean

assert A, B

> (!both A B)

Theorem: (and A B)
```

Conjunction elimination is performed by the two unary methods `left-and` and `right-and`. The former takes a conjunction (and p_1 p_2) and returns p_1, provided that (and p_1 p_2) is in the assumption base (an error occurs otherwise). The method `right-and` behaves likewise, except that it produces the right conjunct instead of the left:

```
clear-assumption-base

assert (A & B)

> (!left-and (A & B))

Theorem: A

> (!right-and (A & B))

Theorem: B

> (!right-and (C & B))

Error, standard input, 1.2: Failed application of right-and---the sentence
(and C B) is not in the assumption base.

> (!left-and (A | B))
```

```
Error, standard input, 1.2: Failed application of left-and---the given sentence
must be a conjunction, but here it was a disjunction: (or A B).
```

Another unary primitive method is dn, which performs double-negation elimination. It takes a premise p of the form (not (not q)), and provided that p is in the assumption base, it returns q:

```
assert p := (~ ~ A)

> (!dn p)

Theorem: A
```

There are several kinds of deductions beyond method applications. In what follows we present a brief survey of the most common deductive forms other than method applications.

2.10.1 Compositions

One of the most fundamental proof mechanisms is *composition*, or sequencing: Assembling a number of deductions $D_1, \ldots, D_n$, $n > 0$, to form the compound deduction

$$\{D_1; \;\cdots\; ; D_n\}. \tag{2.9}$$

To evaluate such a deduction in an assumption base β, we first evaluate D_1 in β. If and when the evaluation of D_1 results in a conclusion p_1, we go on to evaluate D_2 in $\beta \cup \{p_1\}$, that is, in β *augmented with the conclusion of the first deduction*. When we obtain the conclusion p_2 of D_2, we go on to evaluate D_3 in

$$\beta \cup \{p_1, p_2\},$$

namely, in the initial assumption base augmented with the conclusions of the first two deductions; and so on.[20] Thus, each deduction D_{i+1} is evaluated in an assumption base that incorporates the conclusions of all preceding deductions $D_1, \ldots, D_i$. The conclusion of each intermediate deduction thereby serves as a *lemma* that is available to all subsequent deductions. The conclusion of the last deduction D_n is finally returned as the conclusion of the entire sequence. For example, suppose we evaluate the following code in the empty assumption base $\emptyset$:

```
> assert (A & B)

The sentence
(and A B)
has been added to the assumption base.
```

20 Strictly speaking, we should say that D_2 is evaluated *in a copy of* β augmented with p_1; D_3 is evaluated in a copy of β augmented with p_1 and p_2; and so on. We don't statefully (destructively) add each intermediate conclusion to the initial assumption base itself.

```
 6
 7  > {
 8      (!left-and  (A & B));                # This gives A
 9      (!right-and (A & B));                # This gives B
10      (!both B A)                          # And finally, (B & A)
11    }
12
13  Theorem: (and B A)
```

The `assert` directive adds the conjunction `(A & B)` to the initially empty assumption base. Accordingly, the starting assumption base in which the proof sequence on lines 7–11 will be evaluated is $\beta = \{$`(A & B)`$\}$. The proof sequence itself consists of three method calls. The first one, the application of `left-and` to `(A & B)` on line 8, successfully obtains the conclusion `A`, because the required premise `(A & B)` is in the assumption base in which this application is evaluated. Then the second element of the sequence, the application of `right-and` on line 9, is evaluated in the starting assumption base β augmented with the conclusion of the first deduction, that is, in $\{$`(A & B)`,`A`$\}$. That yields the conclusion `B`. Finally, we go on to evaluate the last element of the sequence, the application of `both` on line 10. That application is evaluated in β augmented with the conclusions of the two previous deductions, that is, in

$\{$`(A & B)`,`A`,`B`$\}$.

Since both of its inputs are in the assumption base, `both` happily produces the conclusion `(B & A)`, which thus becomes the conclusion of the entire composition, reported as a theorem on line 13.

At the end of the entire composite proof, only the final conclusion `(B & A)` is retained and added to the starting assumption base β. The intermediate conclusions generated during the evaluation of the composition (namely, the sentences `A` and `B` generated by the `left-and` and `right-and` applications) are *not* retained. For instance, here is what we get if we try to see whether `A` is in the assumption base immediately after the above:

```
> (holds? A)

Term: false
```

In general, whenever we evaluate a deduction D at the top level and obtain a result p:

```
> D

Theorem: p
```

only the conclusion p is added to the global assumption base. Any auxiliary conclusions derived in the course of evaluating D are discarded.

As you might have noticed, when we talk about evaluating a deduction D we often speak of the contents of "the assumption base." For instance, we say that evaluating an application of `left-and` to a sentence `(`p `&` q`)` produces the conclusion p provided that the conjunction `(`p `&` q`)` is in "the assumption base." This does not necessarily refer to the global assumption base that Athena maintains at the top level. Rather, it refers to *the assumption base in which the deduction D is being evaluated*, which may or may not be the global assumption base. For instance, if D happens to be the third member of a composite deduction, then the assumption base in which D will be evaluated will not be the global assumption base; it will be some superset of it. We will continue to speak simply of "the assumption base" in order to keep the exposition simple, but this is an important point to keep in mind.

We refer to a deduction of the form (2.9) as an *inference block*. Each D_i is a *step* of the block. This is not reflected in (2.9), but in fact a step does not have to be a deduction D; it may be an expression E. So, in general, a step of an inference block can be an arbitrary phrase F. The very last step, however, must be a deduction.

Even more generally, a step of an inference block may be named, so that we can refer to the result of that step later on in the block by its given name. A named step is of the form I `:=` F, where I is an identifier or the wildcard pattern (underscore), and F is a phrase. Using this feature, we could express the above proof block as follows:

```
{
  p1 := (!left-and A-and-B);
  p2 := (!right-and A-and-B);
  (!both p2 p1)
}
```

However, we encourage the use of **`let`** deductions instead of inference blocks; we will discuss **`let`** shortly (Section 2.10.3). A **`let`** proof can do anything that an inference block can do, but it is more structured and usually results in more readable code.

2.10.2 Nested method calls

Procedure calls can be nested, that is, the arguments to a procedure can themselves be procedure calls. This is a common feature of all higher-level programming languages, and a style that is especially emphasized in functional languages. Similarly, the arguments to a method can themselves be method calls or other deductions. (Arguments to a method can also be arbitrary expressions.) Suppose that a deduction D appears as an argument to an application of some method M:

$$\texttt{(!}M \ \cdots D \cdots \texttt{)}.$$

To evaluate such a method call in an assumption base β, we first need to evaluate every argument in β; in particular, we will need to evaluate D in β, obtaining from it some conclusion p. Now, when all the arguments have been evaluated and we are finally ready to apply M to their respective values, that application will occur in β *augmented with* p.

Thus, the conclusion of D will be available as a lemma by the time we come to apply M. This is, therefore, a basic mechanism for lemma formation, i.e., for proof composition.

For example, suppose we have defined `conj` as `(A & (B & C))` and consider the deduction

$$\texttt{(!left-and (!right-and conj))} \tag{2.10}$$

in an assumption base β that contains only the premise `conj`:

$$\beta = \{\texttt{(A \& (B \& C))}\}.$$

Here, the deduction `(!right-and conj)` appears directly as an argument to `left-and`. To evaluate (2.10) in β, we begin by evaluating the argument `(!right-and conj)` in β. That will successfully produce the conclusion `(B & C)`, since the required premise is in β. We are now ready to apply the outer method `left-and` to `(B & C)`; but this application will take place in β *augmented with* `(B & C)`, that is, in the assumption base

$$\beta' = \beta \cup \{\texttt{(B \& C)}\} = \{\texttt{(A \& (B \& C))}, \texttt{(B \& C)}\},$$

and hence it will successfully produce the conclusion `B`:

```
clear-assumption-base

assert conj := (A & (B & C))

> (!left-and (!right-and conj))

Theorem: B
```

In general, every time a deduction appears as an argument to a method call, the conclusion of that deduction will appear in the assumption base in which the method will be applied.

2.10.3 Let expressions and deductions

Composing expressions with nested procedure calls is common in the functional style of programming, but Athena also allows for a more structured style using **let** expressions, which let us *name* the results of intermediate computations. Similarly, in addition to composing deductions with nested method calls, Athena also permits **let** deductions, which likewise allow for naming intermediate results. The most common syntax form of the **let** construct is:[21]

$$\texttt{let}\ \{I_1 := F_1; \cdots ; I_n := F_n\}\ F \tag{2.11}$$

where $I_1, \ldots, I_n$ are identifiers and $F_1, \cdots, F_n$ and F are phrases. If F, which is called the *body* of the **let** construct, is an expression, then so is the whole **let** construct. And if the

21 We will see later (in Appendix A) that more general *patterns* can appear in place of the identifiers $I_1, \ldots, I_n$.

body F is a deduction, then the whole **let** construct is also a deduction. So whether or not (2.11) is a deduction depends entirely on whether or not the body F is a deduction.

An expression or deduction of this form is evaluated in a given assumption base β as follows: We first evaluate the phrase F_1. Now, F_1 is either a deduction or an expression. If it is an expression that produces a value V_1, then we just bind the identifier I_1 to V_1 and move on to evaluate the next phrase, F_2, in β. But if F_1 is a deduction that produces some conclusion p_1, we not only bind I_1 to p_1, but we also go on to evaluate the next phrase F_2 in β *augmented with* p_1. We then do the same thing with F_2. If it is an expression, we simply bind I_2 to the value of F_2 and proceed to evaluate F_3; but if it is a deduction, we bind I_2 to the conclusion of F_2, call it p_2, and then move on to evaluate F_3 in β augmented with p_1 *and* p_2. Thus, the conclusion of every intermediate deduction becomes available as a lemma to all subsequent deductions, including the body F when that is a deduction. Moreover, if the conclusion of an intermediate deduction happens to be a conjunction p_i, then all the conjuncts of p_i (and their conjuncts, and so on) are also inserted in the assumption base before moving on to subsequent deductions, and hence they also become available as lemmas.

Here is an example of a **let** expression:

```
> let { a := 1;
        b := (a plus a)
      }
    (b times b)

Term: 4
```

Here, `(a plus a)` is evaluated with `a` bound to `1`, producing `2`; `b` is then bound to `2`, so the body `(b times b)` evaluates to `4`. The entire **let** phrase is an expression because the body `(b times b)` is an expression (a procedure application, specifically).

An example of a **let** deduction is:

```
assert hyp := (male peter & female ann)

> let { left  := (!left-and hyp);
        right := (!right-and hyp)
      }
   (!both right left)

Theorem: (and (female ann)
              (male peter))
```

Here the call to `both` succeeds precisely because it is evaluated in an assumption base that contains the results of the two intermediate deductions—the calls to `left-and` and `right-and`. The entire **let** phrase is a deduction because its body is a deduction (a method application, specifically).

In the expression example, all of the phrases involved are expressions, and in the deduction example all of the phrases are deductions. But any mixture of expressions and deductions is allowed. For example,

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

> let { goal := (B & A);
        _ := (print "Proving: " goal);
        _ := (!right-and hyp);          # this proof step deduces B
        _ := (!left-and hyp) }          # and this one derives A
    (!both B A)

Proving:
(and B A)

Theorem: (and B A)
```

This example also illustrates that when we do not care to give a name to the result of an intermediate phrase F_i, we can use the wildcard pattern _ as the corresponding identifier.

2.10.4 Conclusion-annotated deductions

Sometimes a proof can be made clearer if we announce its intended conclusion ahead of time. This is common in practice. Authors often say "and now we derive p as follows: $\cdots$" or "p follows by $\cdots$." In Athena such annotations can be made with the **conclude** construct, whose syntax is **conclude** p D. Here D is an arbitrary deduction and p is its intended conclusion.

To evaluate a deduction of this form in an assumption base β, we first evaluate D in β. If and when we obtain a conclusion q, we check to ensure that q is the same as the expected conclusion p (up to alpha-equivalence). If so, we simply return p as the final result. If not, we report an error to the effect that the conclusion was different from what was announced:

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

> conclude A
    (!left-and p)

Theorem: A

> conclude B
    (!left-and p)

standard input:1:2: Error: Failed conclusion annotation.
The expected conclusion was:
B
but the obtained result was:
A.
```

In its full generality, the syntax of this construct is **conclude** E D, where E is an arbitrary expression that denotes a sentence. This means that E may spawn an arbitrary amount of computation, as long as it eventually produces a sentence p. We then proceed normally by evaluating D to get a conclusion q and then comparing p and q for alpha-equivalence. In addition, a name I may optionally be given to the conclusion annotation, whose scope becomes the body D:

$$\texttt{conclude}\ I := E\ \ D.$$

This is often useful with top-level uses of **conclude**, when we prove a theorem and define its name at the same time:

```
conclude plus-commutativity :=
  (forall ?x ?y . ?x + ?y = ?y + ?x)
    D
```

2.10.5 Conditional expressions and deductions

In both expressions and deductions, conditional branching is performed with the **check** construct. The syntax of a **check** expression is

$$\texttt{check}\ \{F_1 \Rightarrow E_1\ \mid\ \cdots\ \mid\ F_n \Rightarrow E_n\} \tag{2.12}$$

where the F_i => E_i pairs are the *clauses* of (2.12), with each clause consisting of a *condition* F_i and a corresponding *body* expression E_i. A **check** deduction has the same form, but with deductions D_i as bodies of the clauses:

$$\texttt{check}\ \{F_1 \Rightarrow D_1\ \mid\ \cdots\ \mid\ F_n \Rightarrow D_n\}. \tag{2.13}$$

To evaluate a **check** expression or deduction, we evaluate the conditions $F_1, \ldots, F_n$, in that order. If and when the evaluation of some F_i produces `true`, we evaluate the corresponding body E_i or D_i and return its result as the result of the entire expression or deduction. The last condition, F_n, may be the keyword **else**, which is treated as though it were `true`. It is an error if no F_i produces `true` and there is no **else** clause at the end.

```
assert A

> check {(holds? false) => 1 | (holds? A) => 2 | else => 3}

Term: 2
```

2.10.6 Pattern-matching expressions and deductions

Another form of conditional branching is provided by pattern-matching expressions or deductions. A pattern-matching expression has the form

$$\texttt{match}\ F\ \{\pi_1 \Rightarrow E_1 \ \mid\ \cdots\ \mid\ \pi_n \Rightarrow E_n\} \tag{2.14}$$

where the phrase F is called the *discriminant*, while the π_i => E_i pairs are the *clauses* of (2.14), with each clause consisting of a *pattern* π_i and a corresponding *body* expression E_i. For a description of the various forms of patterns and the details of the pattern-matching algorithm, see Appendix A.4; additional discussion and examples can be found in this chapter in Section 2.11.

The syntax of a pattern-matching deduction is the same, except that the body of each clause must be a deduction:

$$\texttt{match}\ F\ \{\pi_1 \Rightarrow D_1 \ \mid\ \cdots\ \mid\ \pi_n \Rightarrow D_n\}. \tag{2.15}$$

A pattern-matching expression or deduction is evaluated in a given environment ρ and assumption base β as follows. We first evaluate the discriminant F, obtaining from it a value V. We then try to *match* V against the given patterns $\pi_1, \ldots, \pi_n$, in that order. If and when we succeed in matching V against some π_i, resulting in a number of bindings, we go on to evaluate the corresponding body E_i or D_i in ρ augmented with these bindings, and in β.[22] The result produced by that evaluation becomes the result of the entire pattern-matching expression or deduction. An error occurs if the discriminant value V does not match any of the patterns.

```
> match [1 2] {
    [] => 99
  | (list-of h _) => h
  }

Term: 1

> match [1 2] {
    [] => (!claim false)
  | (list-of _ _) => (!true-intro)
  }

Theorem: true
```

22 If the discriminant F is a deduction that produces a conclusion p, and the body is a deduction D_i, then D_i will be evaluated in $\beta \cup \{p\}$. In that case, therefore, the conclusion of the discriminant will serve as a lemma during the evaluation of D_i.

2.10.7 Backtracking expressions and deductions

A form of backtracking is provided by **try** expressions and deductions. A **try** expression has the form

$$\texttt{try}\ \{\ E_1\ \mid\ \ldots\ \mid\ E_n\} \tag{2.16}$$

where $n > 0$. Such an expression does what its name implies: it tries to evaluate each expression E_i in turn, $i = 1, \ldots, n$, until one succeeds, i.e., until some E_i is found that successfully produces a value V_i. At that point V_i is returned as the result of the entire **try** expression. It is an error if all n expressions fail. For example:

```
> try { (4 div 0) | 2 }

Term: 2

> try { (4 div 0) | 25 | (head []) }

Term: 25

> try { (4 div 0) | (head []) }

standard input:1:2: Error: Try expression error; all alternatives failed.
```

A **try** deduction has the same form as (2.16), except that the alternatives are deductions rather than expressions:

$$\texttt{try}\ \{\ D_1\ \mid\ \ldots\ \mid\ D_n\} \tag{2.17}$$

for $n > 0$. For example:

```
> try { (!left-and false) |
        (!true-intro)     |
        (!right-and (true ==> false))}

Theorem: true

> try { (!left-and false) |
        (!right-and (true ==> false))}

standard input:1:2: Error: Try deduction error; all alternatives failed.
```

2.10.8 Defining procedures and methods

Users can define their own procedures with the **lambda** construct, and then use them as if they were primitive procedures. For instance, here is a procedure that computes the square of a given number:[23]

23 For simplicity, we often use "number" synonymously with "numeric term" (i.e., a term of sort `Int` or `Real`).

```
> define square := lambda (n) (n times n)

Procedure square defined.

> square

Procedure: square (defined at standard input:1:32)

> (square 4)

Term: 16
```

The general form of a `lambda` expression is

$$\texttt{lambda}\ (I_1 \cdots I_n)\ E \tag{2.18}$$

where $I_1 \cdots I_n$ are identifiers, called the *formal parameters* of the `lambda` expression, and E is an expression, called the *body* of the `lambda`.

At the top level it is not necessary to define procedures with `lambda`. An alternative notation is the following:

```
> define (square n) := (n times n)

Procedure square defined.
```

or in more traditional Lisp notation:

```
(define (square n)
   (times n n))
```

Any of these alternatives gives the name `square` to the procedure

`lambda (n) (n times n).`

However, it is possible to use the `lambda` construct directly, without giving a name to the procedure it defines. In that case we say that the procedure is *anonymous*. This is particularly useful when we want to simply pass a procedure as an argument to another procedure. For example, the built-in `map` procedure takes a procedure f as its first argument and a list L as its second, and returns the list formed by applying f to each element of L. If $L = [V_1 \cdots V_n]$, then

$$(\texttt{map}\ f\ L) = [(f\ V_1) \cdots (f\ V_n)].$$

For example, since we already defined `square`, we can write

```
> (map square [1 2 3 4 5])

List: [1 4 9 16 25]
```

But we could also pass the squaring procedure to `map` anonymously:

```
> (map lambda (n) (n times n)
       [1 2 3 4 5])

List: [1 4 9 16 25]
```

Defining a procedure is often a process of abstraction: By making an expression the body of a procedure, with some (not necessarily all) of the free identifiers of the expression as formal parameters, we abstract it into a general algorithm that can be applied to other inputs.

Likewise, a given deduction can often be abstracted into a general method that can be applied to other inputs. Consider, for instance, a deduction that derives `(B & A)` from `(A & B)`:

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

It should be clear that there is nothing special here about the atoms `A` and `B`. We can replace them with any other sentences p and q and the reasoning will still go through, as long as the conjunction (p & q) is in the assumption base. So, just like a particular computation such as the squaring of 4 can be abstracted into a general squaring procedure by replacing the constant `4` by a formal parameter like `n`, so we can turn the preceding deduction into a general proof *method* as follows:

```
method (p q)
  let {_ := (!left-and (p & q));
       _ := (!right-and (p & q))}
  (!both q p)
```

This method can be applied to two arbitrary conjuncts p and q, and will produce the conclusion

$$(q \ \& \ p)$$

provided that the premise (p & q) is in the assumption base. While the method could be applied anonymously, it is more convenient to give it a name first:

```
clear-assumption-base

define commute-and :=
  method (p q)
    let {_ := (!left-and (p & q));
         _ := (!right-and (p & q))}
    (!both q p)

assert (B & C)

```

```
11 > (!commute-and B C)
12
13 Theorem: (and C B)
14
15 > (!commute-and A B)
16
17 standard input:3:15: Error: Failed application of left-and---the sentence
18 (and A B) is not in the assumption base.
```

The last example failed because the necessary premise (A & B) was not in the assumption base at the time when the method was applied.

These examples illustrate an important point: When a defined method *M* is called, *M* is applied in the assumption base in which the call takes place, not the assumption base in which *M* was defined.[24] Here, when commute-and was defined (on lines 3–7), the assumption base was empty. But by the time the call on line 11 is made, the assumption base contains exactly one sentence, namely (B & C), so *that* is the logical context in which that application of commute-and takes place. Later, when commute-and is called on line 15, the assumption base contains exactly two sentences, (B & C) as well as (C & B), the theorem produced by the preceding deduction.

A stylistic note: In order to make methods composable, it is preferable, when possible, to define a method *M* so that its only inputs are the premises that it requires. Doing so ensures that *M* can take deductions as arguments, which will serve to establish the required premises prior to the application of *M* (by the semantics of nested method calls, as discussed in Section 2.10.2). Accordingly, the preceding method is better written so that it takes the required conjunction as its sole input, rather than the two individual conjuncts as two separate arguments:

```
define commute-and' :=
  method (premise)
    match premise {
      (p & q) => let {_ := (!left-and premise);
                      _ := (!right-and premise)}
                 (!both q p)
    }
```

This version uses pattern matching to take apart the given premise and retrieve the individual conjuncts, after which the reasoning proceeds as before. The interface and style of commute-and' would normally be preferred over that of commute-and on composability grounds. For instance, suppose the assumption base contains (~ (~ (A & B))) and we want to derive (B & A). Using the second version, we can express the proof in a single line by composing double negation and conjunction commutation:

24 We thus say that method closures have static lexical scoping but *dynamic assumption scoping*.

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

> (!commute-and' (!dn premise))

Theorem: (and B A)
```

Such composition is not possible with the former version.

The same alternative notation that is available for defining procedures can also be used for defining methods: Instead of writing

$$\mathtt{define}\ M\ \mathtt{:=}\ \mathtt{method}\ (I_1 \cdots I_n)\ D$$

we can write

$$\mathtt{define}\ (M\ I_1 \cdots I_n)\ \mathtt{:=}\ D,$$

or, in prefix,

$$(\mathtt{define}\ (M\ I_1 \cdots I_n)\ D).$$

For instance:

```
1 > define (commute-and p q) :=
2    let {_ := (!left-and (p & q));
3         _ := (!right-and (p & q))}
4      (!both q p)
5
6 Method commute-and defined.
```

How does Athena know that this is a method and not a procedure (observe that it responded by acknowledging that a *method* by the name of `commute-and` was defined)? It can tell because the body (lines 2–4) is a deduction. And how can it tell that? In general, how can we tell whether a given phrase F is an expression or a deduction? Recall that expressions and deductions are distinct syntactic categories; there is one grammar for expressions and another for deductions. In most cases, a deduction is indicated just by the leading keyword (the reserved word with which the phrase begins):

`apply-method` (usually abbreviated as `!`)
`assume`
`pick-any`
`pick-witness`
`pick-witnesses`
`generalize-over`
`with-witness`

```
suppose-absurd
conclude
by-induction
datatype-cases
```

(Don't worry if you don't yet recognize some of these; we will explain all of them in due course.) In other cases, when the beginning keyword is **`let`**, **`letrec`**, **`check`**, **`match`**, or **`try`**, it is necessary to peek inside the phrase. A **`let`** or **`letrec`** construct is a deduction if and only if its body is. With a **`check`** or **`match`** phrase we simply look at its first clause body, since the clause bodies must be all deductions or all expressions. With a **`try`** phrase, we look at its first alternative. Finally, any phrase not covered by these rules (such as the unit `()`, or a meta-identifier) is an expression. Thus, the question of whether a given phrase is a deduction or an expression can be mechanically answered with a trivial computation, and in Athena this is done at parsing time.

It is important to become proficient in making that determination, to be able to immediately tell whether a given phrase is an expression or a deduction. Sometimes Athena beginners are asked to write a proof D of some result and end up accidentally writing an expression E instead. Readers are therefore advised to review the above guidelines and then tackle Exercise 2.1 in order to develop this skill.

2.11 More on pattern matching

Recall that the general form of a **`match`** deduction is

$$
\begin{array}{l}
\texttt{match } F \texttt{ \{} \\
\quad \pi_1 \;\texttt{=>}\; D_1 \\
\texttt{|}\; \pi_2 \;\texttt{=>}\; D_2 \\
\qquad \vdots \\
\texttt{|}\; \pi_n \;\texttt{=>}\; D_n \\
\texttt{\}}
\end{array}
$$

where $\pi_1, \ldots, \pi_n$ are *patterns* and $D_1, \ldots, D_n$ are deductions. The phrase F is the discriminant whose value will be matched against the patterns. The pattern-matching algorithm is described in detail in Section A.4, but some informal remarks and a few examples will be useful at this point. We focus on **`match`** deductions here, but what we say will also apply to expressions.