10. *Primitive methods*: A primitive method is an explicitly defined method M whose body is an *expression* E (rather than a deduction, as is the usual requirement for every non-primitive method). M can take as many arguments as it needs, and it can produce any sentence it wants as output—whatever the expression E produces for given arguments. Thus, M becomes part of our trusted computing base and we had better make sure that the results that it produces are justified. Such a method is introduced by the following syntax:

    primitive-method $(M\ I_1 \cdots I_n)$:= E

    where $I_1 \cdots I_n$ are the arguments of M. One can think of Athena's primitive methods as having been introduced by this mechanism. For example, one can think of modus ponens as:

    ```
    primitive-method (mp premise-1 premise-2) :=
      match [premise-1 premise-2] {
        ([(p ==> q) p] where (hold? [premise-1 premise-2])) => q
      }
    ```

    Normally there is no reason to use **primitive-method**, unless we need to introduce infinitely many axioms in one fell swoop, typically as instances of a single axiom schema. In that case a **primitive-method** is the right approach. An illustration (indeed, the only use of this construct in the entire book) is given in Exercise 4.38.

2.17 Summary and notational conventions

Below is a summary of the most important points to take away from this chapter, as well as the typesetting and notational conventions we have laid down:

- Expressions and deductions play fundamentally different roles. Expressions represent arbitrary computations and can result in values of any type whatsoever, whereas deductions represent logical derivations and can only result in sentences. We use the letters E and D to range over the sets of expressions and deductions, respectively.
- A *phrase* is either an expression or a deduction. We use the letter F to range over the set of phrases.
- Expressions and deductions are not just semantically but also syntactically different. Whether a phrase F is an expression or a deduction is immediately evident, often just by inspecting the leading keyword of F.
- Athena keywords (such as **assume**) are displayed in bold font and dark blue color.
- Athena can be used in batch mode or interactively. In interactive mode, if the input typed at the prompt is not syntactically balanced (either a single token or else starting

and ending with parentheses or brackets), then it must be terminated either with a double semicolon `;;` or with `EOF`.

- The syntax of expressions and deductions is defined by mutual recursion. Expressions may contain deductions and all deductions contain expressions.
- All phrases (both expressions and deductions) are evaluated with respect to a given lexical environment ρ, assumption base β, store σ, and symbol set γ. If a deduction D is evaluated with respect to an assumption base β and produces a sentence p, then p is a logical consequence of β. That is the main soundness guarantee provided by Athena.
- Athena *identifiers* are used to give names to values (and also to sorts and modules). They are represented by the letter I (possibly with subscripts, superscripts, etc.). Identifiers can become bound to values either at the top level, with a directive such as **`define`**, or inside a phrase, with a mechanism like **`let`** or via pattern matching inside a **`match`** phrase, and so on.
- Athena *values* are divided into a number of types, enumerated below.[35] Evaluating any phrase F must produce a value of one of these types, unless the evaluation diverges or results in an error:

  1. *Terms*, such as `(father peter)` or `(+ ?x:Int 1)`, which are essentially syntax trees used to represent elements of various sorts. We use the letters s and t to represent terms. *Variables* are a special kind of terms, of the form $?I:S$, where S is a sort. They act as syntactic placeholders. We use the letters x and y to range over variables.
  2. *Sentences*, such as `(= 1 1)`, `(not false)`, etc. We use the letters p, q, and r to represent sentences. Evaluating a deduction can only produce a value of this type.
  3. *Lists* of values, such as `[1 2 [joe (not true)] 'foo]`. We use the letter L to range over lists of values.
  4. The *unit value* `()`.
  5. *Function symbols*, such as `true` or `+`. We use the letters f, g, and h to range over function symbols. Function symbols whose range is `Boolean` are called *relation* or *predicate* symbols; we use the letters P, Q, and R to range over those.
  6. *Sentential constructors* and *quantifiers*, namely `not`, `and`, `or`, `if`, `iff`, `forall`, and `exists`.
  7. *Procedures*, whether they are primitive (such as `plus`) or user-defined via **`lambda`**.[36]
  8. *Methods*, whether they are primitive (such as `both`) or user-defined via **`method`**.

35 The division is not a partition, as some values belong to more than one type. For instance, every Boolean term (such as `true`) is both a term and a sentence.

36 We often use the word "procedure" to refer both to syntactic objects, namely, *expressions* of the form **`lambda`** $(I_1 \cdots I_n)$ E; and to semantic objects, namely, the *denotations* of such expressions, which are proper mathematical functions. Sometimes, when we want to be explicit about the distinction, we speak of a *procedure*

9. ASCII *characters*.
10. *Substitutions*, which are finite mappings from term variables to terms.
11. *Cells* and *vectors*, which can be used to store and to destructively modify arbitrary values or sequences thereof.
12. *Tables* and *maps*, which implement dictionary data types whose keys can be (almost) arbitrary Athena values, the former as hash tables and the latter as functional trees.

We use the letter V to range over Athena values.

- Every term t has a certain *sort*, which may be monomorphic (such as `Ide` or `Boolean`) or polymorphic (such as `'S` or `(Pair 'S1 'S2)`). Sorts are not the same as types. Types, enumerated above, are used to classify Athena values, whereas sorts are used to classify Athena terms, which are just one particular type of value among several others.

2.18 Exercises

Exercise 2.1: Determine whether each of the following phrases is an expression or a deduction. Assume that `A`, `B`, `C`, and `D` have been declared as `Boolean` constants.

```
1. (!both A B)

2. let {p := (A | B)}
     (!both p p)

3. 1700

4. (2 plus 8.75)

5. 'cat

6. let {p := (A & B)}
     match p {
      (_ & _) => (!left-and p)
     }

7. "Hello world!"

8. assume A
     assume B
       (!claim A)

9. [1 2 3]
```

value to refer to such a function. But usually the context will make clear whether we are talking about an expression (syntax) or the abstract function that is the value of such an expression (semantics). Similar remarks apply to our use of the term "method."

```
10. (tail L)

11. lambda (x)
      (x times x times x)

12. lambda (f)
      lambda (g)
        lambda (x)
          (f (g x))

13. let {L := ['a 'b 'c]}
      (rev L)

14. let {x := 2;
         y := 0}
      try { (x div y) | (x times y) }

15. let {p := A;
         q := (B | C)}
      match (p & q) {
        (p1 & (p2 | p3)) => 'match
      | _ => 'fail
      }

16. pick-any x
      (!reflex x)

17. let {g := letrec {fact := lambda (n)
                                check {
                                  (n less? 2) => 1
                                | else => (n times fact n minus 1)
                                }}
                fact}
      (g 5)


18. pick-witness w for (exists ?x . ?x = ?x)
      (!true-intro)

19. check {
       (less? x y) => (!M1)
     | else => assume A (!M 2)
    }

20. letrec {M := method (p)
                   match p {
                      (~ (~ q)) => (!M (!dn p))
                    | _ => (!claim p)
                   }}
```

```
      assume h := (~ ~ ~ ~ A)
       (!M h)

21. let {M := method (p)
               (!both p p)}
      [M 1]
```

Explain your answer in each case. □

Exercise 2.2: Determine the type of the value of each of the following expressions.

1. `2`
2. `true`
3. `(not false)`
4. `[5]`
5. `()`
6. `'a`
7. `+`
8. `(head [father])`
9. `‘A`
10. `or`
11. `lambda (x) x`
12. `|'a := 1|`
13. `(father joe)`
14. `(+ ?x:Int 1)`
15. `"foo"`
16. `make-vector 10 ()`
17. `(match-terms 1 ?x)`
18. `method (p) (!claim (not p))`
19. `(HashTable.table 10)`

If the value is a term, then also state the sort of the term. □

Exercise 2.3: Find a phrase F such that `(!claim` F`)` always succeeds (in every assumption base). □

Exercise 2.4: Find a deduction D that always fails (in every assumption base). □

Exercise 2.5: Implement some of Athena's primitive procedures for manipulating lists, specifically: `map`, `foldl`, `foldr`, `filter`, `filter-out`, `zip`, `take`, `drop`, `for-each`, `for-some`, `from-to`, and `rd`. These are all staples of functional programming, and most of them generalize naturally to data structures other than lists (e.g., to trees and beyond). They are specified as follows:

- `map` takes a unary procedure f and a list of values `[`$V_1 \cdots V_n$`]` and produces the list `[(`$f\ V_1$`)` $\cdots$ `(`$f\ V_1$`)]`.
- `foldl` takes a binary procedure f, an identity element e (typically the left identity of f, i.e., whichever value e is such that $(f\ e\ V) = V$ for all V), and a list of values `[`$V_1 \cdots V_n$`]` and produces the result

$$(f \cdots (f\ (f\ e\ V_1)\ V_2) \cdots V_n).$$

- `foldr` takes a binary procedure f, an identity element e (typically the right identity of f), and a list of values `[`$V_1 \cdots V_n$`]`, and produces the result

$$(f\ V_1\ \cdots (f\ V_{n-1}\ (f\ V_n\ e)) \cdots).$$

- `filter` takes a list L and a unary procedure f that always returns `true` or `false` and produces the sublist of L that contains all and only those elements x of L such that $(f\ x) =$ `true`, listed in the order in which they occur in L (with possible repetitions included).
- `filter-out` works like `filter` except that it only keeps those elements x for which $(f\ x) =$ `false`.
- `zip` is a binary convolution procedure that maps a pair of lists to a list of pairs. Specifically, given two lists `[`$V_1 \cdots V_n$`]` and `[`$V'_1 \cdots V'_m$`]` as arguments, `zip` returns the list `[[`$V_1\ V'_1$`]` $\cdots$ `[`$V_k\ V'_k$`]]`, where k is the minimum of n and m.
- `take` is a binary procedure that takes a list L and an integer numeral n and returns the list formed by the first n elements of L, assuming that n is nonzero and L has at least n elements. If L has fewer than n elements or n is negative, L is returned unchanged. It is an error if n is not an integer numeral.
- `drop` takes a list L and an integer numeral n and returns the list obtained from L by "dropping" the first n elements. If n is not positive then L is returned unchanged, and if n is greater than the length of L, the empty list is returned.
- `for-each` takes a list L and a unary procedure f that always returns `true` or `false`; and returns `true` if $(f\ x)$ is `true` for *every* element x of L, and `false` otherwise.
- `for-some` has the same interface as `for-each` but returns `true` if $(f\ x)$ is `true` for *some* element x of L, and `false` otherwise.

- `from-to` takes two integer numerals a and b and produces the list of all and only those integers i such that $a \leq i \leq b$, listed in numeric order. If $a > b$ then the empty list is returned. This procedure is also known by the infix-friendly name `to`, used as follows:

```
> (5 to 10)

List: [5 6 7 8 9 10]
```

- The `rd` procedure takes a list L and produces the list L' obtained from L by removing all duplicate element occurrences from it, while preserving element order. □

Exercise 2.6: Define a unary procedure `flatten` that takes a list of lists $L_1, \ldots, L_n$ and returns a list of all elements of $L_1, \ldots, L_n$, in the same order in which they appear in the given arguments. For instance, `(flatten [[1 2] [3 4 5]])` should return `[1 2 3 4 5]`. □

Exercise 2.7: Athena's library defines a unary procedure `get-conjuncts` that takes as input a conjunction p and outputs a list of all its nonconjunctive conjuncts, in left-to-right order, where a nonconjunctive conjunct of p is either an immediate subsentence of p that is not itself a conjunction, or else it is a nonconjunctive conjunct of an immediate subsentence of p that is itself a conjunction. If p is not a conjunction, then the single list `[`p`]` is returned. Thus, for example, if the input p is

```
((A & (B | C)) & (D & ~ E) & F)
```

then the result should be `[A (B | C) D (~ E) F]`. Use one of the higher-order list procedures of the previous exercises to implement `get-conjuncts`. Implement a similar procedure `get-disjuncts` for disjunctions. □

Exercise 2.8: Define a ternary procedure `list-replace` that takes (i) a nonempty list of values $L = [V_1 \cdots V_n]$; (ii) a positive integer $i \in \{1, \ldots, n\}$; and (iii) a unary procedure f; and returns the list `[`$V_1 \cdots V_{i-1}$ `(`f V_i`)` $V_{i+1} \cdots V_n$`]`. That is, it returns the list obtained from the input list L by replacing its i^{th} element, V_i, by `(`f V_i`)`. For instance,

```
(list-replace [1 5 10] 2 lambda (n) (n times n))
```

should return `[1 25 10]`. An error should occur if the index i is not in the proper range. □