2 Introduction to Athena

ATHENA is a language for expressing proofs and computations. We begin with a few remarks on computation. As a programming language, Athena is similar to Scheme. It is higher-order, meaning that procedures are first-class values that can be passed as (possibly anonymous) arguments or returned as results; it is dynamically typed, meaning that type checking is performed at run-time; it has side effects, featuring updatable cells and vectors; it relies heavily on lists for structuring data, and those lists can be heterogeneous (containing elements of arbitrary types); and the main tool for control flow is the procedure call.

Nonetheless, Athena's programming language differs from Scheme in several respects. At the somewhat superficial level of concrete syntax, it differs in allowing infix notation in addition to the prefix (s-expression) notation of Scheme. A more significant difference lies in Athena's formal semantics, which are a function of not just the usual semantic abstractions, such as a lexical environment and a store, but also of a novel semantic item called an *assumption base*, which represents, unsurprisingly, a finite set of assumptions. The fact that this new semantic abstraction is built into the formal semantics of the core language is enabled by (and in turn requires) another point of divergence of Athena from more conventional programming languages: differences in the underlying data values, the most salient one being that the concept of a logical proposition, expressed by a formal sentence, takes center stage in Athena.

The basic data values of Scheme are more or less like those of other programming languages: numbers, strings, and so on. From these basic values, Scheme can build other, more complex values, such as lists and procedures. But the fundamental computational mechanisms of Scheme (which are essentially those of the λ-calculus, i.e., mainly procedural abstraction and application) can be deployed on any domain (type) of primitive values, not just the customary ones. The computational mechanisms remain the same, though the universe of primitive values can vary. Athena deploys these same computational mechanisms but on a largely different set of primitive values.

Athena's primitive data values include characters and other such standard fare. But these are not the fundamental data values of the language. The fundamental data values are *terms* and *sentences*. A term is a symbolic structure, essentially a tree whose every node contains either a *function symbol* or a *variable*. Terms should be familiar to you from previous logic courses, or even from elementary math. Expressions such as $3+5$, $x/2$, 78, etc., are all terms. So are names such as *Joe*, expressions such as *Joe's father*, and so on. The job of terms is to *denote* or represent individual objects in some domain of interest. For instance, the term $3+5$ denotes the integer 8, while the term *Joe* presumably designates some individual by that name, and *Joe's father* represents the father of that individual. A *sentence* is essentially a formula of first-order logic: either an atomic formula, or a Boolean combination of formulas, or a quantification. We will describe these in greater detail later in

this chapter. Sentences are used to express propositions about various domains of interest. They serve as the conclusions of proofs.

Computations in Athena usually involve terms and sentences. Procedures, for instance, typically take terms or sentences as inputs, manipulate them in some way or other, and eventually produce some term or sentence as output. Athena does provide numbers (though these are just special kinds of terms), characters, strings, rudimentary I/O facilities, etc., so it can be used as a general-purpose programming language. But the focus is on terms and sentences, as these are the fundamental ingredients for writing proofs. So the programming language of Athena can be viewed as a Scheme-like version of the λ-calculus designed specifically to compute with terms and sentences.[1]

To repeat, there are two uses for Athena: (a) writing programs, usually, though not necessarily, intended to compute with terms and sentences; and (b) writing proofs. Accordingly, there are two fundamental syntactic categories in the language: *deductions*, which represent proofs, and *expressions*, which represent computations (programs).[2] We typically use the letters D and E to range over the sets of all deductions and expressions, respectively. Deductions and expressions are distinct categories, each with its own syntax and semantics, but the two are intertwined: Deductions always contain expressions, and expressions may contain deductions. A *phrase* F is either an expression or a deduction. Thus, if we were to write a BNF grammar describing the syntax of the language, it would look like this:

$$\begin{array}{lll} E & := & \cdots \quad \text{(Expressions, for computing)} \\ D & := & \cdots \quad \text{(Deductions, for proving)} \\ F & := & E \mid D \quad \text{(Phrases)} \end{array} \tag{2.1}$$

Intuitively, the difference between an expression and a deduction is simple. A deduction D represents a logical argument (a proof), and so, if successful, it can only result in one type of value as output: a *sentence*, such as a conjunction or negation, that expresses some proposition or other, say, that all prime numbers greater than 2 are odd. That sentence represents the *conclusion* of the proof. An expression E, by contrast, is not likewise constrained. An expression represents an arbitrary computation, so its output could be a value of any type, such as a numeric term, or an ASCII character, or a list of values, and so on. It may even be a sentence. But there is a crucial difference between sentences produced by expressions versus sentences produced by deductions. A sentence that results from a deduction is guaranteed to be a logical consequence of whatever assumptions were in effect at the time when the deduction was evaluated. No such guarantee is provided for a sentence produced by a computation. A computation can generate any sentence it wishes, including an outright contradiction, without any restrictions whatsoever. But a deduction has to play

1 While both terms and sentences could be represented by appropriate data structures in any general-purpose programming language, having them as built-in primitives woven into the syntactic and semantic kernel of the language carries a number of advantages.

2 In this book, we use the terms “deduction” and “proof” interchangeably.

within a sharply delimited sandbox: It can only result in sentences that are entailed by the assumptions that are currently operative. So the moves that a deduction can make at any given time are much more curtailed.

As with most functionally oriented languages, Athena's mode of operation can be pictured as follows: On one side we have phrases (i.e., expressions or deductions), which are the syntactically well-formed strings of the language; and on the other side we have a class of *values*, such as characters, terms, sentences, and the like. Athena works by evaluating input phrases and displaying the results. This process is called *evaluation*, because it produces the value that corresponds to a given phrase. Evaluation might not always succeed in yielding a value. It might fail to terminate (by getting into an infinite loop), or it might generate an error (such as a division by zero). We write $\boldsymbol{V}$ for the set of all values, and we use the letter V as a variable ranging over $\boldsymbol{V}$.

A phrase F cannot be evaluated in a vacuum. For one thing, the phrase may contain various names, and we need to know what those names mean in order to extract the value of F. Consider, for example, a procedure application such as `(plus x 2)`, where `plus` is a procedure that takes two numeric terms and produces their sum. To compute the value of this expression, we need to know the value of `x`. In general, we need to know the lexical *environment* in which the evaluation is taking place. An environment is just a mapping from names (identifiers) to values. For instance, if we are working in the context of an environment in which the name `x` is bound to `7`, then we can compute the output `9` as the value of `(plus x 2)`. Second, since a phrase may be a proof, and every proof relies on some working assumptions, also known as *premises*, we also need to be given a set of premises—a so-called *assumption base*. Third, since Athena has imperative features such as updatable memory locations ("cells") and vectors, we also need to be given a *store*, which is essentially a function that tells us whether a given memory location is occupied or empty, and if occupied, the value residing there. Finally, we need a *symbol set*, which is a finite collection of sorts and function symbols along with their signatures (we will explain what these are shortly).

In summary, the evaluation of a phrase F always occurs *relative to*, or *with respect to* these four semantic parameters: (a) a lexical environment ρ; (b) an assumption base β; (c) a store σ; and (d) a symbol set γ. Schematically:

$$\text{Input: Phrase } F \xrightarrow[(\text{w.r.t. given } \rho, \beta, \sigma, \gamma)]{\textit{Evaluation}} \text{Output: Value } V$$

A rigorous description of the language would require a precise specification of the syntax of phrases, via a grammar of the form (2.1); a precise specification of the set of values $\boldsymbol{V}$; and a precise description of exactly what value V, if any, corresponds to any phrase F,

relative to a given environment, assumption base, store, and symbol set. Such a description is given in Appendix A. In this chapter we take a less formal approach. We will be chiefly concerned with expressions, that is, with the computational aspects of Athena. We will describe most available kinds of expressions and explain what values they produce, usually without explicit references to the given environment, assumption base, store, or symbol set. We start with terms and sentences in Sections 2.3 and 2.4, respectively. After that we move on to other features of Athena. Deductions will be briefly addressed in general terms in Section 2.10, but the syntax and semantics of most deductions will be introduced in later chapters as needed. Before we can get to terms and sentences, we need to cover some preliminary material on domains and function symbols, in Section 2.2. And before that still, we need to say a few words on how to interact with Athena.

Bear in mind that it is not necessary to go through this chapter from start to finish. Only Sections 2.1–2.10 and the summary (Section 2.17) need to be read in their entirety before continuing to other chapters. (Also, at least the first five exercises are very strongly recommended.) You can return later to the remaining sections on an as-needed basis.

2.1 Interacting with Athena

Athena can be used either in batch mode or interactively. The interactive mode consists of a read-eval-print loop similar to that of other languages (Scheme, ML, Python, Prolog, etc.). The user enters some input in response to the prompt >, Athena evaluates that input, displays the result, and the process is then repeated. The user can quit at any time by entering `quit` at the input prompt.

Typing directly at the prompt all the time is tiresome. It is often more convenient to edit a chunk of code in a file, say `file.ath`, and then have Athena read the contents of that entire file, processing the inputs sequentially from the top of the file to the bottom as if they had been entered directly at the prompt. This can be done with the **`load`** directive. Typing

```
load "file.ath"
```

at the input prompt will process `file.ath` in that manner. The file extension `.ath` can be omitted; to load `foo.ath` it suffices to write **`load`** `"foo"`. Also, when Athena is started, it can be given a file name as a command-line argument, and the file will be loaded into the session.

Note that **`load`** commands can themselves appear inside files. If the command

```
load "file1"
```

is encountered while loading `file.ath`, Athena will proceed to load `file1.ath` and then resume loading the rest of `file.ath`.

Inside a file, the character # marks the beginning of a comment. All subsequent characters until the end of the line are ignored.

A word on code listings: A typical listing displays the input prompt >, followed by some input to Athena, followed by a blank line, followed by Athena's response. Additional Athena inputs, usually preceded by the prompt > and their respective outputs, may follow. If Athena's response to a certain input is immaterial, we omit it, and, in addition, we do not show the prompt before that input. So the absence of the prompt > before some Athena input indicates that the system's response to that input is irrelevant for the purposes at hand and will be omitted. We encourage you to try out all of the code that you encounter as you read this chapter and the ones that follow.

When input is directly entered at the prompt, it may consist of multiple lines. Accordingly, Athena needs to be told when the input is complete, so that it will know to evaluate what has been entered and display the result. This end-of-input condition can be signalled by typing a double semicolon `;;`, or by typing `EOF`, and then pressing Enter. However, this is only necessary if the input is not syntactically balanced. The input is syntactically balanced if it either consists of a single token (a single word or number), or else it starts with a left parenthesis (or bracket) and ends with a matching right parenthesis (or bracket, respectively). For those inputs it is not necessary to explicitly type `;;` or `EOF` at the end—simply press Enter (even if the input consists of multiple lines) and Athena will realize that the input is complete because it has been balanced in the sense we just described, and it will then go on to evaluate what you have typed. But if the input is not syntactically balanced in that way, then `;;` or `EOF` must be typed at the end. Of course, if one is writing Athena code in a file `f.ath` to be loaded later, then these end-of-input markers do not need to appear anywhere inside `f`. In the code listings that follow we generally omit these markers.

2.2 Domains and function symbols

A domain is simply a set of objects that we want to talk about. We can introduce one with the **domain** keyword. For example,

```
> domain Person

New domain Person introduced.
```

introduces a domain whose elements will be (presumably) the persons in some given set.[3] The following is an equivalent way of introducing this domain:

```
(domain Person)
```

3 Whatever set we happen to be concerned with; it could be the set of students registered for some college course, the set of citizens of some country, or the set of all people who have ever lived or ever will live. In general, the *interpretation* of the domains that we introduce is up to us; this point is elaborated in Section 5.6.

Both of the above declarations are syntactically valid. In general, Athena code can be written either in prefix form, using fully parenthesized Lisp-like "s-expressions," or in (largely) infix form that requires considerably fewer parentheses. In this book we generally use infix as the default notation, because we believe that it is more readable, but s-expressions have their own advantages and, for those who prefer them, Appendix A specifies the prefix version of every Athena construct.

Multiple domains can be introduced with the **domains** keyword:

```
domains Element, Set
```

There are no syntactic restrictions on domain names. Any legal Athena identifier *I* can be used as the name of a domain.[4]

Domains are *sorts*. There are other kinds of sorts besides domains (namely *datatypes*, discussed in Section 2.7), but domains are the simplest sorts there are.

Once we have some sorts available we can go ahead and declare *function symbols*, for instance:

```
> declare father: [Person] -> Person

New symbol father declared.
```

This simply says that `father` is a symbol denoting an operation (function) that takes a person and produces another person. We refer to the expression `[Person] -> Person` as the *signature* of `father`. At this point Athena knows nothing about the symbol `father` other than its signature.

The general syntax form for a function symbol declaration is

declare f: [D_1 $\cdots$ D_n] -> D

where f is an identifier and the D_i and D are previously introduced sorts, $n \geq 0$. We refer to $D_1 \cdots D_n$ as the *input sorts* of f, and to D as the *output sort*, or as the *range* of f. The number of input sorts, n, is the *arity* of f. Function symbols must be unique, so they cannot be redeclared at the top level (although a symbol can be freely redeclared inside different modules). There is no conventional overloading whereby one and the same function symbol can be given multiple signatures at the top level, but Athena does provide a different (and in some ways more flexible) form of overloading, described in Section 2.13.

For brevity, multiple function symbols that share the same signature can be declared in a single line by separating them with commas, for instance:

---

4 An identifier in Athena is pretty much any nonempty string of printable ASCII characters that is not a reserved word (Appendix A has a complete list of all reserved words); does not start with a character from this set: {!, ", #, $, ', (,), ,, :, ;, ?, [,], `}; and doesn't contain any characters from this set: {", #,), ,, :, ;,]}. Throughout this book, we use the letter *I* as a variable ranging over the set of identifiers.

```
declare union, intersection: [Set Set] -> Set

declare father, mother: [Person] -> Person
```

A function symbol of arity zero is called a *constant symbol*, or simply a constant. A constant symbol c of domain D can be introduced simply by writing

declare c: D

instead of **declare** c: `[] ->` D. The two forms are equivalent, though the first is simpler and more convenient.

```
> declare joe: Person

New symbol joe declared.

> declare null: [] -> Set

New symbol null declared.
```

Multiple constant symbols of the same sort can be introduced by separating them with commas:

```
declare peter, tom, ann, mary: Person

declare e, e1,e2: Element

declare S, S1, S2: Set
```

There are several built-in constants in Athena. Two of them are `true` and `false`, both of which are elements of the built-in sort `Boolean`. The two numeric domains `Int` (integers) and `Real` (reals) are also built-in. Every integer numeral, such as `3`, `594`, `(- 2)`, etc., is a constant symbol of sort `Int`, while every real numeral (such as `4.6`, `.217`, `(- 1.5)`, etc.) is a constant symbol of sort `Real`.

A function symbol whose range is `Boolean` is also called a *relation* (or *predicate*) symbol, or just "predicate" for short. Some examples:

```
declare in: [Element Set] -> Boolean

declare male, female: [Person] -> Boolean

declare siblings: [Person Person] -> Boolean

declare subset: [Set Set] -> Boolean
```

Here `in` is a binary predicate that takes an element and a set and "returns" `true` or `false`, presumably according to whether or not the given element is a member of the given set.[5] The symbols `male` and `female` are unary predicates on `Person`, while `siblings` and `subset` are binary predicates on `Person` and `Set`, respectively.

Some function symbols are built-in. One of them is the binary equality predicate `=`, which takes any two objects of the same sort S and returns a `Boolean`. S can be arbitrary.[6]

We will generally use the letters f, g, and h as typical function symbols; a, b, c, and d as constant symbols; and P, Q, and R as predicates. The letters f, g, and h will actually do double duty; they will also serve as variables ranging over Athena procedures. The context will always disambiguate their use.

Function symbols are first-class data values in Athena. They can be denoted by identifiers, passed to procedures as arguments, and returned as results of procedure calls.

We stress that function symbols are not procedures. They are ordinary data values that can be manipulated by procedures.[7] An example of a procedure that takes function symbols as arguments is `get-precedence`, which we discuss in the next section.

A procedure is the usual sort of thing written by users, a (possibly recursive) lambda abstraction designed to compute anything from the factorial function to much more complicated functions. Here is an Athena procedure for computing factorials, for example:

```
define (fact n) :=
  check {
    (less? n 1) => 1
  | else => (times n (fact (minus n 1)))
  }
```

where `less?`, `times`, and `minus` are primitive (built-in) procedures operating on numeric terms:

```
> (less? 7 8)

Term: true

> (times 2 3)

Term: 6

> (minus 5 1)
```

5 As with domains, the interpretation of the function symbols that we declare (what these symbols actually *mean*) is up to us. We will soon see how to write down axioms that can prescribe the meaning of a symbol.

6 More precisely, `=` is a *polymorphic* function symbol. We discuss polymorphism in more detail in Section 2.8.

7 Of course, procedures themselves are also "ordinary data values that can be manipulated by procedures," as is the case in every higher-order functional language. But there is an important difference: Procedures cannot be compared for equality, whereas function symbols can. This ostensibly minor technicality has important notational ramifications.

```
Term: 4
```

We will see a few more primitive procedures in this chapter.

A function symbol like `union` or `father`, by contrast, is just a constant data item. We cannot perform any meaningful computations by applying such function symbols to arguments; all we can get by such applications are unreduced terms, as we will see in the next section.

2.3 Terms

A term is a syntactic object that represents an element of some sort. The simplest kind of term is a constant symbol. For instance, assuming the declarations of the previous section, if we type `joe` at the Athena prompt, Athena will recognize the input as a term:

```
> joe

Term: joe
```

We can also ask Athena to print the sort of this (or any other) term:

```
1 > (println (sort-of joe))
2
3 Person
4
5 Unit: ()
```

Athena knows that `joe` denotes an individual in the domain `Person`, so it responds by printing the name of that domain on line 3. (The *unit value* `()` that appears on line 5 is the value returned by the procedure `println`. Most procedures with side effects return the unit value. We will come back to this later.)

A *variable* is also a term. It can be thought of as representing an indeterminate or unspecified individual of some sort, but that is just a vague intuition aid—what exactly is an "unspecified individual" after all? It is more precise to say that a variable is a syntactic object acting as a placeholder that can be replaced by other terms in appropriate contexts. Variables are of the form $?I{:}S$, for any identifier I and any sort S; we refer to I as the *name* of the variable and to S as its sort. Thus, the following are all legal variables:

```
?x:Person
?S25:Set
?foo-bar:Int
?b_1:Boolean
?@sd%&:Real
```

There is no restriction on the length of a variable's name, and very few restrictions on what characters may appear in it, as illustrated by the last example (see also footnote 4). We will use $x, y, z, \ldots$ as metavariables, that is, as variables ranging over the set of Athena variables. Occasionally we might write a metavariable x as $x:S$ to emphasize that its sort is S. Thus, a metavariable $x:S$ ranges over all variables of sort S.

Constant symbols and variables are primitive or *simple* terms, with no internal structure. More complex terms can be formed by applying a function symbol f to n given terms $t_1 \cdots t_n$, where n is the arity of f. Such an application is generally written in prefix form as $(f\ t_1 \cdots t_n)$,[8] though see the remarks below about infix notation for binary and unary function symbols. We say that f is the *root symbol* (or just the "root") of the application, and that $t_1 \cdots t_n$ are its *children*. Thus, the application

```
(father (mother joe))
```

is a legal term. Its root is the symbol `father`, and it has only one child, the term `(mother joe)`. Some more examples of legal terms:

```
(in e S),
(union null S2),
(male (father joe)),
(subset null (union ?X null)).
```

There are two primitive procedures that operate on term applications, `root` and `children`, which respectively return the root symbol of an application and its children (as a list of terms, ordered from left to right). For example:

```
define t := (father (mother joe))

> (root t)

Symbol: father

> (children t)

List: [(mother joe)]
```

Note that constant symbols may also be viewed as applications with no children.[9] Thus:

```
> (root joe)

Symbol: joe
```

8 A more common syntax for such terms with which you may be familiar is $f(t_1, \ldots, t_n)$. When we use conventional mathematical notation we will write terms in that syntax.

9 Indeed, Athena treats `(joe)` and `joe` interchangeably.

```
> (children joe)

List: []
```

It is convenient to be able to use these procedures on variables as well. Applying `root` and `children` to a variable x will return x and the empty list, respectively.

We will have more to say about lists later on, but some brief remarks are in order here. A list of $n \geq 0$ values $V_1 \cdots V_n$ can be formed simply by enclosing the values inside square brackets: $[V_1 \cdots V_n]$. For instance:

```
> [tom ann]

List: [tom ann]

> []

List: []

> [tom [peter mary] ann]

List: [tom [peter mary] ann]
```

Note that Athena lists, like those of Scheme but unlike those of ML or Haskell, can be heterogeneous: They may contain elements of different types. The most convenient and elegant way to manipulate lists is via pattern matching, which we discuss later. Outside of pattern matching, there are a number of primitive procedures for working with lists, the most useful of which are `add`, `head`, `tail`, `length`, `rev`, and `join`. The first one is a binary procedure that simply takes a value V and a list L and produces the list obtained by prepending V to the front of L. The functionality of the remaining five procedures should be clear from their names. Note that the concatenation procedure `join` can take an arbitrary number of arguments (one of the few Athena procedures without a fixed arity). Some examples:

```
> (add 1 [2 3])

List: [1 2 3]

> (head [1 2 3])

Term: 1

> (tail [1 2 3])

List: [2 3]

> (rev [1 2 3])

List: [3 2 1]
```

```
> (length [1 2 3])

Term: 3

> (join [1 2] ['a 'b] [3])

List: [1 2 'a 'b 3]
```

The infix-friendly identifier `added-to` is globally defined as an alias for `add`, so we can write, for example:

```
> (rev 1 added-to [2])

List: [2 1]
```

Returning to terms, we say that a term t is *ground* iff it does not contain any variables. Alternatively, t is ground iff it only contains function symbols (including constants). Thus, `(father joe)` is ground, but `(father ?x:Person)` is not.

We generally use the letters s and t (possibly with subscripts/superscripts) to designate terms.

The *subterm* relation can be inductively defined through a number of simple rules, namely:

1. Every term t is a subterm of itself.
2. If t is an application $(f\ \ t_1 \cdots t_n)$, then every subterm of a child t_i is also a subterm of t.
3. A term s is a subterm of a term t if and only if it can be shown to be so by the above rules.

Inductive definitions like this one can readily be turned into recursive Athena procedures, and we will see such examples throughout the book. A *proper* subterm of t is any subterm of t other than itself. If t_3 is a proper subterm of t_1 and there is no term t_2 such that t_2 is a proper subterm of t_1 and t_3 is a proper subterm of t_2, we say that t_3 is an *immediate* subterm of t_1.

As should be expected, every legal term is of a certain sort. The sort of `(father joe)`, for example, is `Person`; the sort of

```
(in ?x:Element ?s:Set)
```

is `Boolean`; and so on. In general, the sort of a term $(f\ \ t_1 \cdots t_n)$ is D, where D is the range of f. Some terms, however, are *ill-sorted*, for example, `(father true)`. An attempt to construct this term would result in a "sort error" because the function symbol `father` requires an argument of sort `Person` and we are giving it a `Boolean` instead. Athena performs sort checking automatically and will detect and report any ill-sorted terms as errors:

```
1 > (father true)
2
3 standard input:1:2: Error: Unable to infer a sort for the term:(father true)
4
5 (Failed to unify the sorts Boolean and Person.)
```

Here `1:2` indicates line 1, column 2, the precise position of the offending term in the input stream; and `standard input` means that the term was entered directly at the Athena prompt. If the term `(father true)` had been read in batch mode from a file `foo.ath` instead, and the term was found, say, on line 279, column 35, the message would have been: `foo.ath:279:35: Error: Unable to infer ···`. Athena prints out the ill-sorted term and gives a parenthetical indication (line 5 above) of why it was unable to infer a sort for it.

Note that we speak of "ill-sorted" terms, "sort errors," and so on, rather than ill-typed terms or type errors. There is an important distinction between types and sorts in Athena. Only terms have sorts, but terms are just one type of value in Athena. There are several other types: sentences, lists, procedures, methods, the unit value, and more. So we say that the type of `null` is term and its sort is `Set`. This crucial distinction will become clearer as you gain more experience with the language.

It is not necessary to annotate every variable occurrence inside a term with a sort. Athena can usually infer the proper sort of every variable occurrence automatically. For instance:

```
> (in ?x ?S)

Term: (in ?x:Element ?S:Set)
```

Here we typed in the term `(in ?x ?S)` without any sort annotations. As we can tell from its response, Athena inferred that in this term the sort of `?x` must be `Element` while that of `?S` is `Set`. Another example:

```
> (father ?p)

Term: (father ?p:Person)
```

Omitting explicit sort annotations for variables can save us a good deal of typing, and we will generally follow that practice, especially when the omitted sorts are easily deducible from the context. Occasionally, however, inserting such annotations can make our code more readable.

A binary function symbol can always be used in infix, that is, between its two arguments rather than in front of them. For example, instead of `(union null S)` we may write

```
(null union S).
```

Also, when we have successive applications of unary function symbols, it is not necessary to enclose each such application within parentheses. Thus, instead of writing

```
(father (mother joe))
```

we may simply write `(father mother joe)`. Accordingly, some of the terms we have seen so far can also be written as follows:

```
(e in S),
(null union S2),
(male father joe),
(null subset ?x union null).
```

Even though users may enter terms in infix, Athena always prints output terms in full prefix form, as so-called "s-expressions":

```
> (null union ?s)

Term: (union null ?s:Set)
```

Prefix notation is generally better for output because it is easier to indent, and proper indentation can be invaluable when trying to read large terms (or sentences, as we will soon see). Note also that when Athena prints its output, all variables are fully annotated with their most general possible sorts. This is the default; it can be turned off by issuing the top-level directive **`set-flag`** `print-var-sorts "off"`; see also Section 2.12.

Every function symbol has a *precedence* level, which can be obtained with the primitive procedure `get-precedence`:

```
> (get-precedence subset)

Term: 100

> (get-precedence union)

Term: 110
```

By default, every binary relation symbol (i.e., binary function symbol with `Boolean` range) is given a precedence of 100, while other binary or unary function symbols are given a precedence of 110.

Precedence levels can be set explicitly with the directive **`set-precedence`**:

```
> set-precedence union 200

OK.
```

Now that `union` has a higher precedence than `intersection` (which has the default 110), we expect an expression such as

```
(?s1 union ?s2 intersection ?s3)
```

to be parsed as `(intersection (union ?s1 ?s2) ?s3)`, and we see that this is indeed the case:

```
> (?s1 union ?s2 intersection ?s3)

Term: (intersection (union ?s1:Set ?s2:Set)
                    ?s3:Set)
```

Binary function symbols also have associativities that can be programmatically obtained and modified. By default, every binary function symbol associates to the right:

```
> (?s1 union ?s2 union ?s3)

Term: (union ?s1:Set
             (union ?s2:Set ?s3:Set))
```

The associativity of a binary function symbol can be explicitly specified with the directives **`left-assoc`** and **`right-assoc`**:

```
> left-assoc union

OK.

> (?s1 union ?s2 union ?s3)

Term: (union (union ?s1:Set ?s2:Set)
             ?s3:Set)
```

The unary procedure `get-assoc` returns a string representing the associativity of its input function symbol.

Infix notation can be used not just for function symbols, but for binary procedures as well. For instance:

```
> (7 less? 8)

Term: true

> (2 times 3)

Term: 6

> (5 minus 1)

Term: 4
```

Thus, the earlier factorial example could also be written as follows:

```
define (fact n) :=
  check {
    (n less? 1) => 1
  | else => (n times fact n minus 1)
  }
```

2.4 Sentences

Sentences are the bread and butter of Athena. Every successful proof derives a unique sentence. Sentences express propositions about relationships that might hold among elements of various sorts. There are three kinds of sentences:

1. Atomic sentences, or just *atoms*. These are simply terms of sort `Boolean`. Examples are

   ```
   (siblings peter (father joe))
   ```

   and `(subset ?s1 (union ?s1 ?s2))`.

2. Boolean combinations, obtained from other sentences through one of the five *sentential constructors*:[10] `not`, `and`, `or`, `if`, and `iff`, or their synonyms, as shown in the following table.

Constructor	Synonym	Prefix mode	Infix mode	Interpretation
`not`	`~`	`(not` p`)`	`(~` p`)`	Negation
`and`	`&`	`(and` p q`)`	`(`p `&` q`)`	Conjunction
`or`	`|`	`(or` p q`)`	`(`p `|` q`)`	Disjunction
`if`	`==>`	`(if` p q`)`	`(`p `==>` q`)`	Conditional
`iff`	`<==>`	`(iff` p q`)`	`(`p `<==>` q`)`	Biconditional

   One can use any connective or its synonym in either prefix or infix mode, although, by convention, the synonyms are typically used in infix mode. As with terms, Athena always writes output sentences in prefix. In infix mode, precedence levels come into play, often allowing the omission of parenthesis pairs that would be required in prefix mode. The procedure `get-precedence` can be used to inspect the precedence levels of these connectives, but briefly, conjunction binds tighter than disjunction, and both bind tighter than conditionals and biconditionals. Negation has the highest precedence of all five sentential constructors. Therefore, for example, `(~ A & B | C)` is understood as the

10 We use the term *sentential* (or "logical") *connective* interchangeably with "sentential constructor." Note that the term "constructor" is not used here in the technical datatype sense introduced in Section 2.7.

disjunction of `((~ A) & B)` and `C`. All binary sentential constructors associate to the right by default. (This can be changed with the **`left-assoc`** directive, but such a change is not recommended.) In a conjunction (*p* `&` *q*), we refer to *p* and *q* as *conjuncts*, and in a disjunction (*p* `|` *q*), we refer to *p* and *q* as *disjuncts*. A conditional (*p* `==>` *q*) can also be read as "if *p* then *q*" or even as "*p* implies *q*."[11] We call *p* the *antecedent* and *q* the *consequent* of the conditional. We will see that a biconditional (*p* `<==>` *q*) is essentially the same as the conjunction of the two conditionals (*p* `==>` *q*) and (*q* `==>` *p*), and for that reason it is sometimes called an *equivalence*, as it states that *p* and *q* are equivalent (insofar as each implies the other).

The combination of negation with equality has its own shorthand:

Procedure	Synonym	Prefix	Infix	Interpretation
`unequal`	`=/=`	(`unequal` *s t*)	(*s* `=/=` *t*)	*s* is not equal to *t*

Because function symbols and sentential constructors are regular data values, users can define their own abbreviations and use them in either prefix or infix mode. For instance:

```
define \/ := or
```

When written in prefix, conjunctions and disjunctions can be polyadic; that is, the sentential constructors `and` and `or` may receive an arbitrary number of sentences as arguments, not just two. We will have more to say on that point shortly.

3. Quantified sentences. There are two quantifiers in Athena, `forall` and `exists`. A quantified sentence is of the form (*Q* *x* : *S* `.` *p*) where *Q* is a quantifier, *x* : *S* is a variable of sort *S*, and *p* is a sentence—the *body* of the quantification, representing the *scope* of the quantified variable *x* : *S*. The period is best omitted if the body *p* is written in prefix, but, when present, it must be surrounded by spaces. For example:

```
(forall ?x . ?x =/= father ?x).
```
(2.2)

Intuitively, a sentence of the form (`forall` *x* : *S* `.` *p*) says that *p* holds for every element of sort *S*, while (`exists` *x* : *S* `.` *p*) says that there is some element of sort *S* for which *p* holds. In summary:

11 It is debatable whether the sentential constructor ==> has any meaningful correspondence to the informal notion of implication. Many logicians have argued that implication between two propositions requires strong relationships that cannot be captured by any truth-functional semantics (like the semantics of ==> that we present in Section 4.12). For the purposes of this book, however, such debates may be ignored, so we will freely "speak with the vulgar" and refer to conditionals as implications.

Quantifier	Prefix	Infix	Interpretation
`forall`	`(forall` x`:`S p`)`	`(forall` x`:`S `.` p`)`	p holds for every x:S
`exists`	`(exists` x`:`S p`)`	`(exists` x`:`S `.` p`)`	p holds for some x:S

We generally use the letters p, q, and r as variables ranging over sentences.

The following shows what happens when we type sentence (2.2) at the Athena prompt:

```
> (forall ?x . ?x =/= father ?x)

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))
```

Athena acknowledges that a sentence was entered, and displays that sentence using proper indentation. Note that we did not have to explicitly indicate the sort of the quantified variable. Athena inferred that `?x` ranges over `Person`, and annotated every occurrence of `?x` with that sort. But, had we wanted, we could have explicitly annotated the quantified variable with its sort:

```
> (forall ?x:Person . ?x =/= father ?x)

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))
```

Or we could annotate every variable occurrence with its sort, or only some such occurrences:

```
> (forall ?x:Person . ?x:Person =/= father ?x:Person)

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))

> (forall ?x . ?x =/= father ?x:Person)

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))
```

It is usually best to omit as many variable annotations as possible and let Athena work them out.

Because the body of (2.2) is written entirely in infix, we did not have to insert any additional parentheses in it. Had we wanted, we could have entered the sentence in a more explicit form by parenthesizing either the `(father ?x)` term or the entire body of the quantified sentence:

```
> (forall ?x:Person . ?x:Person =/= (father ?x:Person))

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))
```

or:

```
> (forall ?x:Person . (?x:Person =/= (father ?x:Person)))

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))
```

Typically, however, when we write sentences in infix we omit as many pairs of parentheses as possible. In full prefix notation, we would write sentence (2.2) as `(forall ?x (=/= ?x (father ?x)))`. Note that in prefix we typically omit the dot immediately before the body of the quantified sentence.

As a shorthand for iterated occurrences of the same quantifier, one can enter sentences of the form $(Q\ x_1 \cdots x_n\ .\ p)$ for a quantifier Q and $n > 0$ variables $x_1 \cdots x_n$. For instance:

```
> (forall ?S1 ?S2 . ?S1 = ?S2 <==> ?S1 subset ?S2  & ?S2 subset ?S1)

Sentence: (forall ?S1:Set
            (forall ?S2:Set
              (iff (= ?S1:Set ?S2:Set)
                   (and (subset ?S1:Set ?S2:Set)
                        (subset ?S2:Set ?S1:Set)))))
```

Another shorthand exists for prefix applications of `and` and `or`. Both of these can take an arbitrary number $n > 0$ of sentential arguments: `(and` $p_1 \cdots p_n$`)` and `(or` $p_1 \cdots p_n$`)`. In infix mode we can get somewhat similar results with (p_1 `&` $\cdots$ `&` p_n) and (p_1 `|` $\cdots$ `|` p_n), respectively. Note, however, that these two latter expressions will give nested and right-associated applications of the binary versions of `and` and `or`, whereas in the case of `(and` $p_1 \cdots p_n$`)` or `(or` $p_1 \cdots p_n$`)` we have a *single* sentential constructor applied to an arbitrarily large number of $n > 0$ sentences.

In addition, all five sentential constructors can accept a *list* of sentences as their arguments, which is quite useful for programmatic applications of these constructors:

```
> (and [A B C])

Sentence: (and A
               B
               C)

> (if [A B])
```

```
Sentence: (if A B)

> (not [A])

Sentence: (not A)
```

We inductively define the *subsentence* relation as follows:

1. Every sentence p is a subsentence of itself.
2. If p is a negation (`~` q), then every subsentence of q is also a subsentence of p.
3. If p is a conjunction (or disjunction), then any subsentence of any conjunct (or disjunct) is a subsentence of p.
4. If p is a conditional, then every subsentence of the antecedent or consequent is a subsentence of p.
5. If p is a biconditional (`(`p_1 `<==>` p_2`)`, then every subsentence of p_1 or p_2 is a subsentence of p.
6. If p is a quantified sentence, then every subsentence of the body is a subsentence of p.
7. A sentence q is a subsentence of p if and only if it can be shown to be so by the preceding rules.

As was the case with subterms, this definition too can be easily turned into a recursive Athena procedure; see exercise 3.36. A *proper subsentence* of p is any subsentence of p other than itself. If r is a proper subsentence of p and there is no sentence q such that q is a proper subsentence of p and r is a proper subsentence of q, then we say that r is an *immediate* subsentence of p.

A variable occurrence x`:`S inside a sentence p is said to be *bound* iff it is within a quantified subsentence of p of the form (`(`Q x`:`S `.` p'`)`, for some quantifier Q. A variable occurrence that is not bound is said to be *free*. Free variable occurrences must have consistent sorts across a given sentence. For instance,

```
(?x = joe & male ?x)
```

is legal because both free variable occurrences of `?x` are of the same sort (`Person`), but

```
(male ?x & ?x = 3)
```

is not, because one free occurrence of `?x` has sort `Person` and the other has sort `Int`:

```
> (?x = joe & male ?x)

Sentence: (and (= ?x:Person joe)
               (male ?x:Person))

```

```
6  > (male ?x & ?x = 3)
7
8  Error, top level, 1.1: Unable to verify that this sentence is well-sorted:
9  (and (male ?x:Person)
10      (= ?x:Int 3))
11
12 (Could not satisfy the constraint 'T189 = (Int /\ Person)
13 because Int and Person do not have a g.l.b.)
```

(You can ignore the parenthetical message on lines 12 and 13.) However, bound variable occurrences can have different sorts in the same sentence:

```
> ((forall ?x . male father ?x) & (forall ?x . ?x subset ?x))

Sentence: (and (forall ?x:Person
                 (male (father ?x:Person)))
               (forall ?x:Set
                 (subset ?x:Set ?x:Set)))
```

Intuitively, that is because the name of a bound variable is immaterial. For instance, there is no substantial difference between `(forall ?x . male father ?x)` and

`(forall ?p . male father ?p)`.

Both sentences say the exact same thing, even though one of them uses `?x` as a bound variable and the other uses `?p`. We say that the two sentences are *alpha-equivalent*, or alpha-convertible, or "alphabetic" variants. This means that each can be obtained from the other by consistently renaming bound variables. Alpha-equivalent sentences are essentially identical, and indeed we will see later that Athena treats them as such for deductive purposes. Alpha-conversion, and specifically *alpha-renaming*, occurs automatically very often during the evaluation of Athena proofs. A sentence is alpha-renamed when all bound variable occurrences in it are consistently replaced by fresh (hitherto unseen) variables ranging over the same sorts.

A very common operation is the safe replacement of every free occurrence of a variable v inside a sentence p by some term t. By "safe" we mean that, if necessary, the operation alpha-renames p to avoid any *variable capture* that might come about as a result of the replacement. Variable capture is what would occur if any variables in t were to become accidentally bound by quantifiers inside p as a result of carrying out the replacement. For instance, if p is

`(exists ?x:Real . ?x:Real > ?y:Real)`,

v is `?y:Real`, and t is `(2 * ?x:Real)`, then naively replacing free occurrences of v inside p by t will result in

`(exists ?x:Real . ?x:Real > 2 * ?x:Real)`,

where the third (and boxed) occurrence of `?x:Real` has been accidentally "captured" by the leading bound occurrence of `?x:Real`. An acceptable way of carrying out this replacement is to first alpha-rename p, obtaining an alphabetic variant p' of it, say

$$p' = \texttt{(exists ?w:Real . ?w:Real > ?y:Real)},$$

and then carry out the replacement inside p' instead, obtaining the result

```
(exists ?w:Real . ?w:Real > 2 * ?x:Real).
```

This is legitimate because p and p' are essentially identical. We write $\{v \mapsto t\}(p)$ for this operation of safely replacing all free occurrences of v inside p by t.

We close this section with a tabular summary of Athena's prefix and infix notations for first-order logic sentences vis-à-vis corresponding syntax forms often used in conventional mathematical writing:

	Athena prefix	Athena infix	Conventional
Atoms:	(*R* $t_1 \cdots t_n$)	(t_1 *R* t_2) when $n = 2$	$R(t_1, \ldots, t_n)$ (t_1 *R* t_2)
Negations:	(not *p*)	(~ *p*)	$(\neg p)$
Conjunctions:	(and p_1 p_2)	(p_1 & p_2)	$(p_1 \wedge p_2)$
Disjunctions:	(or p_1 p_2)	(p_1 | p_2)	$(p_1 \vee p_2)$
Conditionals:	(if p_1 p_2)	(p_1 ==> p_2)	$(p_1 \Rightarrow p_2)$
Biconditionals:	(iff p_1 p_2)	(p_1 <==> p_2)	$(p_1 \Leftrightarrow p_2)$
Un. quantifications:	(forall x:S *p*)	(forall x:S . *p*)	$(\forall x{:}S \,.\, p)$
Ex. quantifications:	(exists x:S *p*)	(exists x:S . *p*)	$(\exists x{:}S \,.\, p)$

Occasionally we may use conventional notation, if we believe that it can simplify or shorten the exposition.

2.5 Definitions

Definitions let us give a name to a value and then subsequently refer to the value by that name. This is a key tool for managing complexity. Athena provides a few naming mechanisms. One of the most useful is the top-level directive **`define`**. Its general syntax form is:

`define` *I* := *F*

where I is any identifier and F is a phrase denoting the value that we want to define. Once a definition has been made, the defined value can be referred to by its name:

```
> define p := (forall ?s . ?s subset ?s)

Sentence p defined.

> (p & p)

Sentence: (and (forall ?s:Set
                 (subset ?s:Set ?s:Set))
               (forall ?s:Set
                 (subset ?s:Set ?s:Set)))
```

Athena has static scoping (also known as *lexical scoping*), which means that new definitions override older ones:

```
> define t := joe

Term t defined.

> t

Term: joe

> define t := 0

Term t defined.

> t

Term: 0
```

Because function symbols are regular denotable values, we can easily define names for them and use them as alternative notations. For instance, suppose we want to use the symbol `\/` as an alias for `union` in a given stretch of text. We can do that with a simple definition:

```
define \/ := union
```

We can now go ahead and use `\/` to build terms and sentences and so on:

```
> \/

Symbol: union

> (e in null \/ S)

Term: (in e
          (union null S))
```

Because logical connectives and quantifiers are also regular denotable values, we can do the same thing with them. For instance, if we prefer to use `all` as a universal quantifier, we can simply say:

```
> define all := forall

Quantifier all defined.

> (all ?s . ?s subset ?s)

Sentence: (forall ?s:Set
            (subset ?s:Set ?s:Set))
```

The ability to compute with function symbols (as well as logical connectives and quantifiers) has some distinct advantages. It enables a flexible form of overloading and other powerful notational customizations. These are features that we will be using extensively in this book.

Keep in mind that **`define`** is a top-level directive, not a procedure. It can only appear as a direct command to Athena; it cannot appear inside a phrase.

We can use square-bracket list notation to define several values at the same time. For instance:

```
> define [x y z] := [1 2 3]

Term x defined.

Term y defined.

Term z defined.

> [z y x]

List: [3 2 1]
```

This feature is similar to the use of patterns inside **`let`** bindings, discussed in Section 2.16.

2.6 Assumption bases

At all times Athena maintains a global set of sentences called the *assumption base*. We can think of the elements of the assumption base as our premises—sentences that we regard (at least provisionally) as true. Initially the system starts with a small assumption base. Every time an axiom is postulated or a theorem is proved at the top level,[12] the corresponding sentence is inserted into the assumption base.

12 A theorem is simply a sentence produced by a deduction D. We say that the deduction proves or derives the corresponding theorem (which is also the deduction's *conclusion*).

The assumption base can be inspected with the procedure `show-assumption-base`. It is a nullary procedure (i.e., it does not take any arguments), so it is invoked as

```
(show-assumption-base).
```

A related nullary procedure, `get-ab`, returns a list of all the sentences in the current assumption base. Because this is a very commonly used procedure, it also goes by the simpler name `ab`.

When Athena is first started, the assumption base has a fairly small number of sentences in it, mostly axioms of some very fundamental built-in theories, such as pairs and options.

```
> (length (ab))

Term: 20
```

A sentence can be inserted into the assumption base with the top-level directive **`assert`**:

```
> assert (joe =/= father joe)

The sentence
(not (= joe
        (father joe)))
has been added to the assumption base.
```

The unary procedure `holds?` can be used to determine whether a sentence is in the current assumption base. We can confirm that the previous sentence was inserted into the assumption base by checking that it holds afterward:

```
> (holds? (joe =/= father joe))

Term: true
```

There is a similar procedure `hold?` that takes a list of sentences and returns `true` if all of them are in the assumption base and `false` otherwise.

Multiple sentences can be asserted at the same time; the general syntax form is

`assert` $p_1, \ldots, p_n$

Lists of sentences can also appear as arguments to **`assert`**. A list of sentences `[`$p_1 \cdots p_n$`]` can be given a name with a definition, and then asserted by that name. This is often the most convenient way to make a group of assertions:

```
define facts := [(1 = 1) (joe = joe)]

assert facts
```

Alternatively, we can combine definition with assertion in one fell swoop:

```
assert facts := [(1 = 1) (joe = joe)]
```

All of these features work with **assert*** as well, which is equivalent to **assert** except that it first universally quantifies each given sentence over all of its free variables before inserting it into the assumption base (see page 7).

There are also two mechanisms for removing sentences from the global assumption base:

clear-assumption-base

and **retract**. The former will empty the assumption base, while the second will only remove the specified sentence:

```
> clear-assumption-base

Assumption base cleared.

> assert (1 = 2)

The sentence
(= 1 2)
has been added to the assumption base.

> retract (1 = 2)

The sentence
(= 1 2)
has been removed from the assumption base.
```

All three of these (**assert**, **clear-assumption-base**, and **retract**) are top-level directives; they cannot appear inside other syntax forms.

The last two constructs are powerful and should be used sparingly. A directive

retract *p*,

in particular, simply removes *p* from the assumption base without checking to see what other sentences might have been derived from *p* in the interim (between its assertion and its retraction), so a careless removal may well leave the assumption base in an incorrect state. This tool is meant to be used only when we assert a sentence *p* and then right away realize that something went wrong, for instance, that *p* contains certain free variables that it should not contain, in which case we can promptly remove it from the assumption base with **retract**.

2.7 Datatypes

A *datatype* is a special kind of domain. It is special in that it is *inductively generated*, meaning that every element of the domain can be built up in a finite number of steps by applying

certain operations known as the *constructors* of the datatype. A datatype D is specified by giving its name, possibly followed by some sort parameters (if D is polymorphic; we discuss that in Section 2.8), and then a nonempty sequence of constructor profiles separated by the symbol `|`. A constructor profile without selectors[13] is of the form

$$(c \;\; S_1 \cdots S_n), \tag{2.3}$$

consisting of the name of the constructor, c, along with n sorts $S_1 \cdots S_n$, where S_i is the sort of the i^{th} argument of c. The range of c is not explicitly mentioned—it is tacitly understood to be the datatype D. That is why, after all, c is called a "constructor" of D; because it builds or *constructs* elements of the datatype. Thus, every application of c to n arguments of the appropriate sorts produces an element of D. In terms of the **declare** directive that we have already seen, a constructor c of a datatype D with profile (2.3) can be thought of as a function symbol with the following signature declaration:

declare c: [$S_1 \cdots S_n$] -> D.

A nullary constructor (one that takes no arguments, so that $n = 0$) is called a *constant constructor* of D, or simply a constant of D. Such a constructor represents an individual element of D. The outer parentheses in (2.3) may be (and usually are) omitted from the profile of a constant constructor.

Here is an example:

```
datatype Boolean := true | false
```

This defines a datatype by the name of `Boolean` that has two constant constructors, `true` and `false`. (This particular datatype is predefined in Athena.) Thus, this definition says that the datatype `Boolean` has two elements, `true` and `false`. The definition conveys more information than that, however. It also licenses the conclusion that `true` and `false` are *distinct* elements, and, moreover, that `true` and `false` are the *only* elements of `Boolean`.

The intended effect of this datatype definition could be approximated in terms of mechanisms with which we are already familiar as follows:

```
domain Boolean

declare true, false: Boolean

assert (true =/= false)

assert (forall ?b:Boolean . ?b = true | ?b = false)
```

Here we have made two assertions that ensure that (a) `true` and `false` refer to distinct objects; and (b) the domain `Boolean` does not contain any other elements besides those

13 Selectors are discussed in Section A.5.

denoted by `true` and `false`. For readers who are familiar with universal algebra, these axioms hold because the datatype is freely generated, meaning that it is a *free algebra*. Given a datatype D, we will collectively refer to these axioms as the *free-generation axioms* for D. Roughly, the axioms express the following propositions: (a) different constructor applications create different elements of D; and (b) every element of D is represented by some constructor application. Axioms of the first kind capture what are known as *no-confusion* conditions, while axioms of the second kind capture a limited form of a so-called *no-junk* condition.

Thus, to a certain extent, a datatype definition can be viewed as syntax sugar for a domain declaration (plus appropriate symbol declarations for the constructors), along with the relevant free-generation axioms. We say "to a certain extent" because an additional and important effect of a datatype definition is the introduction of an inductive principle for performing structural induction on the datatype (we discuss this in Section 3.8). For infinite datatypes such as the natural numbers, to which we will turn next, structural induction is an ingredient that goes above and beyond the free-generation axioms, meaning that the induction principle allows us to prove results that are not derivable from the free-generation axioms alone.

Here is a datatype for the natural numbers that we will use extensively in this book:

```
datatype N := zero | (S N)
```

This defines a datatype `N` that has one constant constructor `zero` and one unary constructor `S`. Because the argument of `S` is of the same sort as its result, `N`, we say that `S` is a *reflexive* constructor. Thus, `S` requires an element of `N` as input in order to construct another such element as output. By contrast, `zero`, as well as `true` and `false` in the `Boolean` example, are *irreflexive* constructors—trivially, since they take no arguments.

Unlike domains, which can be interpreted by arbitrary sets, the interpretation of a datatype is fixed: A datatype definition always picks out a unique set (up to isomorphism). Roughly, in the case of `N`, that set can be understood as given by the following rules:

1. `zero` is an element of `N`.
2. For all n, if n is an element of `N`, then `(S` n`)` is an element of `N`.
3. Nothing else is an element of `N`.

The last clause, in particular, ensures the minimality of the defined set, whereby the *only* elements of `N` are those that can be obtained by the first two clauses, namely, *by a finite number of constructor applications*. This is what ensures that the specified set does not have any extraneous ("junk") elements.

Essentially, every datatype definition can be understood as a recursive set definition of the preceding form. How do we know that a recursive definition of this form always succeeds in determining a unique set, and how do we know that a datatype definition can

always be thus understood? We will not get into the details here, but briefly, the answer to the first question is that we can prove the existence of a unique set satisfying the recursive definition using fixed-point theory; and the answer to the second question is that there are simple syntactic constraints on datatype definitions that ensure that every datatype gives rise to a recursive set definition of the proper form.[14]

The following are the free-generation axioms for `N`:

```
(forall ?n . zero =/= S ?n)

(forall ?n ?m . S ?n = S ?m ==> ?n = ?m)

(forall ?n . ?n = zero | exists ?m . ?n = S ?m)
```

You might recognize these if you have ever seen Peano's axioms. The first two are no-confusion axioms: (1) `zero` is different from every application of `S` (i.e., `zero` is not the successor of any natural number), and (2) applications of `S` to different arguments produce different results (i.e., `S` is injective). The third axiom says that the constructors `zero` and `S` span the domain `N`: Every natural number is either `zero` or else the successor of some natural number.[15]

The free-generation axioms of a datatype can be obtained automatically by the unary procedure `datatype-axioms`, which takes the name of the datatype as an input string and returns a list of the free-generation axioms for that datatype:

```
> (datatype-axioms "N")

List: [
(forall ?y1:N
  (not (= zero
          (S ?y1:N))))

(forall ?x1:N
  (forall ?y1:N
    (iff (= (S ?x1:N)
            (S ?y1:N))
         (= ?x1:N ?y1:N))))

(forall ?v:N
  (or (= ?v:N zero)
      (exists ?x1:N
        (= ?v:N
           (S ?x1:N)))))
]
```

14 For instance, definitions such as **`datatype`** `D := (c D)` are automatically rejected.

15 Strictly speaking, the third axiom (and other axioms of a similar form, in the case of other datatypes) can be proved by induction, but it is useful to have it directly available.

```
> (datatype-axioms "Boolean")

List: [
(not (= true false))

(forall ?v:Boolean
  (or (= ?v:Boolean true)
      (= ?v:Boolean false)))
]
```

Whether or not these axioms are added to the assumption base automatically is determined by a global flag `auto-assert-dt-axioms`, off by default. If turned on (see Section 2.12), then every time a new datatype T is defined, its axioms will be automatically added to the assumption base. In addition, the identifier T`-axioms` will be automatically bound in the global environment to the list of these axioms.

Once a datatype D has been defined, it can be used as a bona fide Athena sort; for example, we can declare functions that take elements of D as arguments or return elements of D as results. We can introduce binary addition on natural numbers, for instance, as follows:

```
> declare Plus: [N N] -> N

New symbol Plus declared.
```

A number of mutually recursive datatypes can be defined with the **`datatypes`** keyword, separating the component datatypes with `&&`. For example:

```
> datatypes Even := zero | (s1 Odd) &&
            Odd := (s2 Even)

New datatypes Even and Odd defined.
```

The constructors of a top-level datatype (or a set of mutually recursive datatypes) must have distinct names from one another, as well as from the constructors of every other top-level datatype and from every other function symbol declared at the top level. However, the same sort, constructor, or function symbol name can be used inside different modules.

Not all inductively defined sets are freely generated. For example, in some cases it is possible for two distinct constructor terms to denote the same element. Consider, for instance, a hypothetical datatype for finite sets of integers:

```
datatype Set := null | (insert Int Set)
```

This definition says that a finite set of integers is either `null` (the empty set) or else of the form `(insert` *i* *s*`)`, obtained by inserting an integer *i* into the set *s*. Now, two sets are identical iff they have the same members, so that $\{1,3\} = \{3,1\}$. Hence,

```
(insert 3 (insert 1 null))
```

and

```
(insert 1 (insert 3 null))
```

ought to be identical. That is, we must be able to prove

```
((insert 3 (insert 1 null)) = (insert 1 (insert 3 null))).
```
(2.4)

But one of the no-confusion axioms would have us conclude that `insert` is injective:

```
(forall ?i1 ?s1 ?i2 ?s2 . (insert ?i1 ?s1) = (insert ?i2 ?s2)
                ==> ?i1 = ?i2 & ?s1 = ?s2).
```

This is inconsistent with (2.4), as it would allow us to conclude, for example, that `(1 = 3)`.

The preferred approach here is to define `Set` as a **`structure`** rather than a **`datatype`**:

```
structure Set := null | (insert Int Set)
```

A *structure* is a datatype with a coarser identity relation. Just like a regular datatype, a structure is inductively generated by its constructors, meaning that every element of the structure is obtainable by a finite number of constructor applications. This means that structural induction (via **`by-induction`**) is available for structures. The only difference is that there may be some "confusion," for instance, the constructors might not be injective (the usual case), so that one and the same constructor applied to two distinct sequences of arguments might result in the same value. More rarely, we might even obtain the same value by applying two distinct constructors. It is the user's responsibility to assert a proper identity relation for a structure. In the set example, we would presumably assert something along the following lines:

```
(forall ?s1 ?s2 . ?s1 = ?s2 <==> ?s1 subset ?s2 & ?s2 subset ?s1),
```

where `subset` has the usual definition in terms of membership.

The unary procedure `structure-axioms` will return a list of all the inductive axioms that are usually appropriate for a structure, assuming that the only difference is that constructors are not injective. The input argument is the name of the structure:

```
> (structure-axioms "Set")

List: [
(forall ?y1:Int
  (forall ?y2:Set
    (not (= null
            (insert ?y1:Int ?y2:Set)))))

(forall ?v:Set
  (or (= ?v:Set null)
      (exists ?x1:Int
        (exists ?x2:Set
          (= ?v:Set
             (insert ?x1:Int ?x2:Set))))))
]
```

If the structure has other differences (most notably, if applying two distinct constructors might result in the same value), then it is the user's responsibility to assert the relevant axioms as needed.

2.8 Polymorphism

2.8.1 Polymorphic domains and sort identity

A domain can be polymorphic. As an example, consider sets over an arbitrary universe, call it `S`:

```
> domain (Set S)

New domain Set introduced.
```

The syntax for introducing polymorphic domains is the same as before, except that now the domain name is flanked by an opening parenthesis to its left and a list of identifiers to its right, followed by a closing parenthesis; say, `(Set S)`, or in the general case, $(I\ I_1 \cdots I_n)$, where I is the domain name. The identifiers $I_1, \ldots, I_n$ are parameters that serve as *sort variables* in this context, indicating that I is a *sort constructor* that takes any n sorts $S_1, \ldots, S_n$ as arguments and produces a new sort as a result, namely $(I\ S_1 \cdots S_n)$. For instance, `Set` is a unary sort constructor that can be applied to an arbitrary sort, say the domain `Int`, to produce the sort `(Set Int)`. For uniformity, monomorphic sorts such as `Person` and `N` can be regarded as nullary sort constructors. Polymorphic datatypes and structures, discussed in Section 2.8.3, can also serve as sort constructors.

Equipped with the notion of a sort constructor, we can define Athena sorts more precisely. Suppose we have a collection of sort constructors $\boldsymbol{SC}$, with each $sc \in \boldsymbol{SC}$ having a unique arity $n \geq 0$, and a disjoint collection of sort variables $\boldsymbol{SV}$. We then define a *sort over* $\boldsymbol{SC}$ *and* $\boldsymbol{SV}$ as follows:

- Every sort variable in ***SV*** is a sort over ***SC*** and ***SV***.
- Every nullary sort constructor $sc \in \boldsymbol{SC}$ is a sort over ***SC*** and ***SV***.
- If $S_1, \ldots, S_n$ are sorts over ***SC*** and ***SV***, $n > 0$, and $sc \in \boldsymbol{SC}$ is a sort constructor of arity n, then $(sc\ \ S_1 \cdots S_n)$ is a sort over ***SC*** and ***SV***.
- Nothing else is a sort over ***SC*** and ***SV***.

We write $Sorts(\boldsymbol{SC}, \boldsymbol{SV})$ for the set of all sorts over ***SC*** and ***SV***.

For example, suppose that $\boldsymbol{SC} = \{\texttt{Int}, \texttt{Boolean}, \texttt{Set}\}$ and $\boldsymbol{SV} = \{\texttt{S1}, \texttt{S2}\}$, where `Int` and `Boolean` are nullary sort constructors, while `Set` is unary. Then the following are all sorts over ***SC*** and ***SV***:

```
Boolean,
Int,
(Set Boolean),
S1,
(Set Int),
(Set S2),
(Set (Set Int)),
(Set (Set S1)).
```

There are infinitely many sorts over these three sort constructors and two sort variables.

Sorts of the form $(sc\ \ S_1 \cdots S_n)$ for $n > 0$ are called *compound*, or complex. A *ground* (or *monomorphic*) sort is one that contains no sort variables. All of the above examples are ground except `S1`, `(Set S2)`, and `(Set (Set S1))`. A sort that is not ground is said to be polymorphic.

A polymorphic domain is really a domain *template*. Substituting ground sorts for its sort variables gives rise to a specific domain, which is an *instance* of the template. Intuitively, you can think of a polymorphic domain as the collection of all its ground instances.

Formally, let us define a *sort valuation* τ as a function from sort variables to sorts. Any such τ can be extended to a function

$$\widehat{\tau} : Sorts(\boldsymbol{SC}, \boldsymbol{SV}) \rightarrow Sorts(\boldsymbol{SC}, \boldsymbol{SV})$$

(i.e., to a function $\widehat{\tau}$ from sorts over ***SC*** and ***SV*** to sorts over ***SC*** and ***SV***) as follows:

$$\widehat{\tau}(\psi) = \tau(\psi);$$
$$\widehat{\tau}((sc\ \ S_1 \cdots S_n)) = (sc\ \ \widehat{\tau}(S_1) \cdots \widehat{\tau}(S_n));$$

for any sort variable $\psi \in \boldsymbol{SV}$ and sort constructor $sc \in \boldsymbol{SC}$ of arity n. We say that a sort S_1 is an *instance of* (or *matches*) a sort S_2 iff there exists a sort valuation τ such that $\widehat{\tau}(S_2) = S_1$. And we say that two sorts S_1 and S_2 are *unifiable* iff there exists a sort valuation τ such that $\widehat{\tau}(S_1) = \widehat{\tau}(S_2)$. There are algorithms for determining whether one sort

matches another, or whether two sorts are unifiable, and these algorithms are widely used in Athena's operational semantics.

Two sorts are considered identical iff they differ only in their variable names, that is, iff each can be obtained from the other by (consistently) renaming sort variables. It follows that a ground sort (such as `N`) is identical only to itself, since it does not contain any sort variables.

2.8.2 Polymorphic function symbols

Polymorphic sorts pave the way for polymorphic function symbols. The syntax for declaring a polymorphic function symbol f is the same as before, except that the relevant sort variables must appear listed within parentheses and separated by commas before the list of input sorts. Thus, the general syntax form is

$$\texttt{declare}\ f\texttt{: (}I_1,\ldots,I_n\texttt{) [}S_1\cdots S_n\texttt{] -> }S \tag{2.5}$$

where $I_1,\ldots,I_n$ are distinct identifiers that will serve as sort variables in the context of this declaration, and as before, S_i is the sort of the i^{th} argument and S is the sort of the result. Presumably, some of these sorts will involve the sort variables. For example:

```
declare in: (S) [S (Set S)] -> Boolean

declare union: (S) [(Set S) (Set S)] -> (Set S)

declare =: (S) [S S] -> Boolean
```

The first declaration introduces a polymorphic membership predicate that takes an element of an arbitrary sort S and a set over S and yields either `true` or `false`. The second declaration introduces a polymorphic union function that takes two sets over an arbitrary sort S and produces another set over the same sort. Finally, the last declaration introduces a polymorphic equality predicate; that particular symbol is built-in.

Polymorphic constants can be declared as well, say:

```
declare empty-set: (S) [] -> (Set S)
```

A function symbol declaration of the form (2.5) is admissible iff (a) f is distinct from every function symbol (including datatype constructors) introduced before it; (b) every S_i, as well as S, is a sort over $\boldsymbol{SC}$ and $\{I_1,\ldots,I_n\}$, where $\boldsymbol{SC}$ is the set of sort constructors available prior to the declaration of f; and (c) every sort variable that appears in the output sort S appears in some input sort S_i. Hence, the following are all inadmissible:

```
> declare g: (S) [(Set Int S)] -> Boolean

standard input:1:18: Error: Ill-formed sort: (Set Int S).
```

```
> declare g: (S) [(Set T) S] -> Int

standard input:1:18: Error: Ill-formed sort: (Set T).

> declare g: (S) [Int] -> S

standard input:1:25: Error: The sort variable S appears in the
resulting sort but not in any argument sort.
```

The first declaration is rejected because `(Set Int S)` is not a legal sort (we introduced `Set` as a unary sort constructor, but here we tried to apply it to *two* sorts). The second attempt is rejected because `T` is neither a previously introduced sort nor one of the sort variables listed before the input sorts. The third error message explains why the last attempt is also rejected.

Intuitively, a polymorphic function symbol f can be thought of as a collection of monomorphic function symbols, each of which can be viewed as an instance of f. The declaration of each of those instances is obtainable from the declaration of f by consistently replacing sort variables by ground sorts. For example, the foregoing declaration of the polymorphic predicate symbol `in` might be regarded as syntax sugar for infinitely many monomorphic function symbol declarations:

```
declare in_Int: [Int (Set Int)] -> Boolean

declare in_Real: [Real (Set Real)] -> Boolean

declare in_Boolean: [Boolean (Set Boolean)] -> Boolean

declare in_(Set Int): [(Set Int) (Set (Set Int))] -> Boolean
```

and so on for infinitely more ground sorts. This is elaborated further in Section 5.6.

Polymorphic function symbols are harnessed by Athena's polymorphic terms and sentences. Try typing a variable such as `?x` into Athena without any sort annotations:

```
> ?x

Term: ?x:'T175
```

Note the sort that Athena has assigned to the input variable, namely, `'T175`. This is a sort variable. Athena generally prints out sort variables in the format `'T`n or `'S`n, where n is some integer index. This is the most general sort that could be assigned to the variable `?x` in this context. So this is a polymorphic variable, and hence a polymorphic term. Users can also enter explicitly polymorphic variables, that is, variables annotated with polymorphic sorts, writing sort variables in the form `'`I. For example:

```
> ?y:'T3

Term: ?y:'T177
```

Note that the sort that Athena assigned to `?y` is `'T177`, which is identical to `'T3`, since each can be obtained from the other by renaming sort variables (recall our discussion of sort identity in the previous subsection). Here are some more examples of polymorphic terms:

```
> (?x in ?y)

Term: (in ?x:'T203
          ?y:(Set 'T203))

> (?a = ?b)

Term: (= ?a:'T206 ?b:'T206)

> (?x:(Set 'T) in ?y:(Set (Set 'T)))

Term: (in ?x:(Set 'T209)
          ?y:(Set (Set 'T209)))
```

In the first two examples, Athena automatically infers the most general possible polymorphic sorts for every variable occurrence. Also note that the common occurrence of `'T203` in the first example indicates that Athena has inferred a constraint on the sorts of `?x` and `?y`, namely, that whatever the sort of `?x` is, `?y` must be a set of elements of *that* sort. Likewise for the second example: The sorts of `?a` and `?b` can be arbitrary but they must be identical. In the third example we have explicitly provided specific polymorphic sorts for `?x` and `?y`: `(Set 'T)` for `?x` and `(Set (Set 'T))` for `?y`. Athena verified that these were consistent with the signature of `in` and thus accepted them (modulo the sort-variable permutation `'T` $\leftrightarrow$ `'T209`). Observe, however, that it is not necessary to provide both of these sorts. We can just annotate one of them and have the sort of the other be inferred automatically:

```
> (?x:(Set 'T) in ?y)

Term: (in ?x:(Set 'T213)
          ?y:(Set (Set 'T213)))
```

We can write sort annotations not just for variables but also for constant terms:

```
> (in ?x empty-set:(Set Real))

Term: (in ?x:Real
          empty-set:(Set Real))

> empty-set:(Set (Set 'S))

Term: empty-set:(Set (Set 'T4395))
```

Any constant term can be annotated in Athena, including monomorphic ones. Athena will just ignore such annotations, as long as they are correct:

```
> joe:Person

Term: joe

> (S zero:N)

Term: (S zero)
```

There is an easy way to check whether a term t is polymorphic: Give it as input to the Athena prompt and then scan Athena's output for sort variables (prefixed by `'`). If Athena annotates at least one variable or constant symbol in t with a polymorphic sort (i.e., one containing sort variables), then t is polymorphic; otherwise it is monomorphic. But there is also a primitive unary procedure `poly?` that will take any term t and return `true` or `false` depending on whether or not t is polymorphic.

Informally, it is helpful to think of a polymorphic term as a schema or template that represents an entire set of (possibly infinitely many) monomorphic terms. For instance, you can think of the polymorphic term `empty-set` as representing infinitely many monomorphic terms, such as

```
empty-set:(Set Int),
empty-set:(Set Boolean),
empty-set:(Set (Set Int)),
```

and so on. Keep in mind that the mere presence of a polymorphic symbol in a term does not make that term polymorphic. For instance, the term `(?x:Int = 3)` contains the polymorphic symbol `=`, but it is not itself polymorphic. No variable or constant symbol in it has a nonground sort, as you can verify by typing the term into Athena:

```
> (?x = 3)

Term: (= ?x:Int 3)
```

or by supplying it as an argument to `poly?`:

```
> (poly? (?x = 3))

Term: false
```

The presence of polymorphic function symbols is a necessary condition for a term to be polymorphic, but it is not sufficient.

A polymorphic sentence is one that contains at least one polymorphic term, or a quantified variable with a nonground sort. Here are some examples:

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

Sentence: (forall ?x:'S
            (= ?x:'S ?x:'S))

> (forall ?x ?y . ?x union ?y = ?y union ?x)

Sentence: (forall ?x:(Set 'S)
            (forall ?y:(Set 'S)
              (= (union ?x:(Set 'S)
                        ?y:(Set 'S))
                 (union ?y:(Set 'S)
                        ?x:(Set 'S)))))

> (~ exists ?x . ?x in empty-set)

Sentence: (not (exists ?x:'S
                 (in ?x:'S
                         empty-set:(Set 'S))))
```

Note that quantified variables can be explicitly annotated with polymorphic sorts:

```
> (exists ?x:(Set (Set 'T)) . ?x =/= empty-set)

Sentence: (exists ?x:(Set (Set 'S))
            (not (= ?x:(Set (Set 'S))
                    empty-set:(Set (Set 'S)))))
```

As with terms, a simple way to test whether an unannotated sentence is polymorphic is to give it as input to the Athena prompt and then scan the output for sort variables. If you see any, the sentence is polymorphic, otherwise it is monomorphic. But `poly?` can also be used on sentences:

```
> (poly? (forall ?x . ?x = ?x))

Term: true
```

Also as with terms, a polymorphic sentence such as `(forall ?x . ?x = ?x)` can be seen as a collection of (potentially infinitely many) monomorphic sentences, namely:

```
(forall ?x:Int . ?x = ?x),
(forall ?x:Boolean . ?x = ?x),
(forall ?x:(Set Int) . ?x = ?x),
```

and so on. This expressivity is the power of parametric polymorphism. A single polymorphic sentence can express infinitely many propositions about infinitely many sets of objects.

2.8.3 Polymorphic datatypes

Since datatypes are just special kinds of domains, they too can be polymorphic, and likewise for structures. Athena's polymorphic datatypes resemble polymorphic algebraic datatypes in languages such as ML and Haskell. Here are two examples for polymorphic lists and ordered pairs:

```
datatype (List S) := nil | (:: S (List S))

datatype (Pair S T) := (pair S T)
```

The syntax is the same as for monomorphic datatypes, except that the datatype name is now flanked by an opening parenthesis to its left and a list of distinct identifiers to its right, followed by a closing parenthesis, as in `(List S)` or `(Pair S T)`, or, in general, $(I\ I_1 \cdots I_n)$, where I is the name of the datatype. Just as for polymorphic domains, the identifiers $I_1, \ldots, I_n$ serve as local sort variables, indicating that I is a sort constructor that takes n sorts $S_1, \ldots, S_n$ as arguments and produces a new sort as a result, $(I\ S_1 \cdots S_n)$. For instance, `List` is a unary sort constructor that can be applied to an arbitrary sort, say the domain `Int`, to produce the sort `(List Int)`; while `Pair` is a binary sort constructor and can thus be applied to any two sorts to produce a new one, say, `(Pair Boolean (List Int))`.

The profile of each constructor is the same as before: $(c\ S_1 \cdots S_k)$, where c is the name of the constructor.[16] Here, each S_i must be a sort over all previously introduced sort constructors *plus* the datatype that is being defined (thus allowing recursion), and the sort variables $I_1, \ldots, I_n$. Recursive datatype definitions are quite useful and common, the definition of lists above being a typical example.

The notion of a reflexive constructor remains unchanged: If an argument of a constructor c is of a sort that involves the name of the datatype of which c is a constructor, then c is reflexive; otherwise it is irreflexive. Every datatype must have at least one irreflexive constructor. More specifically, we say that a definition of a datatype D is admissible iff (1) the name D is distinct from all previously introduced sorts (domains or datatypes); (2) the constructors of D are distinct from one another, as well as from every function symbol introduced prior to the definition of D; (3) every argument sort of every constructor of D is a sort over $\boldsymbol{SC} \cup \{D\}$ and $\{I_1, \ldots, I_n\}$, where $\boldsymbol{SC}$ is the set of previously available sort constructors and $I_1, \ldots, I_n$ are the sort variables (if any) listed in the definition of D; and finally, (4) D has at least one irreflexive constructor.

The procedures `datatype-axioms` and `structure-axioms` work just as well with polymorphic datatypes, for example:

```
> (datatype-axioms "List")

List: [
```

16 Also as before, the outer parentheses may be dropped when $k = 0$.

```
(forall ?y1:'S
  (forall ?y2:(List 'S)
    (not (= nil:(List 'S)
            (:: ?y1:'S
                ?y2:(List 'S))))))

(forall ?x1:'S
  (forall ?x2:(List 'S)
    (forall ?y1:'S
      (forall ?y2:(List 'S)
        (if (= (:: ?x1:'S
                   ?x2:(List 'S))
               (:: ?y1:'S
                   ?y2:(List 'S)))
            (and (= ?x1:'S ?y1:'S)
                 (= ?x2:(List 'S)
                    ?y2:(List 'S)))))))))

(forall ?v:(List 'S)
  (or (= ?v:(List 'S)
         nil:(List 'S))
      (exists ?x1:'S
        (exists ?x2:(List 'S)
          (= ?v:(List 'S)
             (:: ?x1:'S
                 ?x2:(List 'S)))))))
]
```

When D is a polymorphic datatype of arity n and $S_1, \ldots, S_n$ are ground sorts, the sort $(D\ S_1 \cdots S_n)$ may be regarded as a monomorphic datatype. That datatype's definition can be retrieved from the definition of D by consistently replacing the sort variables $I_1, \ldots, I_n$ by the sorts $S_1, \ldots, S_n$, respectively. For example, `(Pair Int Boolean)` may be seen as a monomorphic datatype $\texttt{Pair}_{\texttt{Int}\times\texttt{Boolean}}$ with one binary constructor

$$\texttt{pair}_{\texttt{Int}\times\texttt{Boolean}}$$

whose first argument is `Int` and whose second argument is `Boolean`, namely, as the datatype

$$\textbf{datatype}\ \texttt{Pair}_{\texttt{Int}\times\texttt{Boolean}}\ \texttt{:=}\ (\texttt{pair}_{\texttt{Int}\times\texttt{Boolean}}\ \texttt{Int}\ \texttt{Boolean}).$$

Likewise, `(List Int)` can be understood as a monomorphic datatype $\texttt{List}_{\texttt{Int}}$, definable as

$$\textbf{datatype}\ \texttt{List}_{\texttt{Int}}\ \texttt{:=}\ \texttt{nil}_{\texttt{Int}}\ \texttt{|}\ (\texttt{::}_{\texttt{Int}}\ \texttt{Int}\ \texttt{List}_{\texttt{Int}}).$$

Thus, just as with polymorphic domains, a polymorphic datatype may be viewed as the collection of all its ground instances.

2.8.4 Integers and reals

Athena comes with two predefined numeric domains, `Int` for integers and `Real` for real numbers. The domain `Int` has infinitely many constant symbols associated with it, namely, all integer numerals, positive, negative, and zero:

```
> (?x = 47)

Term: (= ?x:Int 47)
```

Note that Athena automatically recognized the sort of `?x` as `Int`. Negative integer numerals are written as `(- n)`:

```
> (exists ?x . ?x = (- 5))

Sentence: (exists ?x:Int
            (= ?x:Int
               (- 5)))
```

Real numerals are written in the usual format, with dots embedded in them to mark the fractional part of the number:

```
> 3.14

Term: 3.14

> (?x = .158)

Term: (= ?x:Real 0.158)
```

There are five predeclared binary function symbols that take numeric terms as arguments: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder). They are overloaded so that they can be used both with integers and with reals, or indeed with any combination thereof:

```
> (?x + 2)

Term: (+ ?x:Int 2)

> (2.3 * ?x)

Term: (* 2.3 ?x:Real)
```

These symbols adhere to the usual precedence and associativity conventions:

```
> (2 * 7 + 35)

Term: (+ (* 2 7)
         35)
```

The subtraction symbol can be used both with one and with two arguments:

```
> (- 2)

Term: (- 2)

> (7 - 5)

Term: (- 7 5)
```

As a unary symbol it represents integer/real negation, and as a binary symbol it represents subtraction. Likewise, + can be used both as a unary and as a binary symbol.

There are also function symbols for the usual comparison operators: < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to). They are likewise overloaded:

```
> (2 <= 5)

Term: (<= 2 5)

> (forall ?x . ?x + 1 > ?x)

Sentence: (forall ?x:Int
            (> (+ ?x:Int 1)
               ?x:Int))
```

Note that if we wanted `?x` to range over `Real` in the last sentence, we could annotate the quantified variable:

```
> (forall ?x:Real . ?x + 1 > ?x)

Sentence: (forall ?x:Real
            (> (+ ?x:Real 1)
               ?x:Real))
```

Alternatively, we could just write the numeral 1 in real-number format, and Athena would then automatically treat `?x` as ranging over `Real`:

```
> (forall ?x . ?x + 1.0 > ?x)

Sentence: (forall ?x:Real
            (> (+ ?x:Real 1.0)
               ?x:Real))
```

There are also predefined procedures for performing the usual computations with numbers: `plus`, `minus`, `times`, `div`, `mod`, `less?`, `greater?`, and `equal?`. We stress that these are not function symbols, that is, they do not make terms but actually perform the underlying computations:

```
> (1 plus 2 times 3)

Term: 7

> (100 div 2)

Term: 50

> (3.0 div 1.5 equal? 2)

Term: true
```

The procedure `equal?` (also defined as `equals?`) is a generic equality test that can be applied to any two values, not just numbers.

2.9 Meta-identifiers

We have already seen that several sorts are predefined in Athena. These include domains such as `Int` and `Real`, as well as datatypes such as `Boolean`. Another built-in domain is `Ide`, the domain of *meta-identifiers*. There are infinitely many constants predeclared for `Ide`. These are all of the form `'`*I*, where *I* is a regular Athena identifier. For example:

```
'foo
'x
'233
'*
'sd8838jd@!
```

These are called meta-identifiers.[17] Examples:

```
> 'x

Term: 'x

> (println (sort-of 'x))

Ide

Unit: ()

> (exists ?x . ?x = 'foo)
```

17 These should not be confused with sort variables. Both have printed representations that start with the quote symbol `'`, but they are very different kinds of things. A meta-identifier such as `'foo` is an Athena *term*. A sort variable such as `'S35` represents a sort, and sorts are what terms have. The sort of every meta-identifier is `Ide`.

```
Sentence: (exists ?x:Ide
            (= ?x:Ide 'foo))
```

Among many other uses, meta-identifiers can represent the variables of some object language whose abstract syntax needs to be modeled by an Athena datatype. As a brief example, consider the untyped λ-calculus, where an expression is either a variable (identifier), or an abstraction, or an application. That abstract grammar could be represented by the following Athena datatype:

```
datatype Exp := (Var Ide) | (Lambda Ide Exp) | (App Exp Exp)
```

Then the term

```
(Lambda 'x (Var 'x))
```

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 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 for 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

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.

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—such as a procedure or a method, but 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.

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, 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:

```
1  > assert (A & B)
2
3  The sentence
4  (and A B)
5  has been added to the assumption base.
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

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

Since both of its inputs are in that 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

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.

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 $(p \texttt{ \& } q)$ produces the conclusion p provided that the conjunction $(p \texttt{ \& } q)$ 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 & B));
  p2 := (!right-and (A & 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:

$$(!M \ \cdots D \cdots).$$

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 proof composition.

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

```
(!left-and (!right-and conj))
```
(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 form of the **let** construct is:[21]

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

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** phrase, is an expression, then so is the whole **let** phrase. And if the body F is a deduction, then the whole **let** 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 into 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:

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

```
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:

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

This is often useful with top-level uses of **conclude**, when we prove a theorem and give it a 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

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

where the $F_i \Rightarrow E_i$ 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 clause bodies:

$$\textbf{check}\ \{F_1 \Rightarrow D_1 \mid \cdots \mid F_n \Rightarrow D_n\}.$$

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

$$\textbf{match}\ F\ \{\pi_1 \Rightarrow E_1\ |\ \cdots\ |\ \pi_n \Rightarrow E_n\} \tag{2.13}$$

where the phrase F is called the *discriminant*, while the $\pi_i \Rightarrow E_i$ pairs are the *clauses* of (2.13), 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 Section 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:

$$\textbf{match}\ F\ \{\pi_1 \Rightarrow D_1\ |\ \cdots\ |\ \pi_n \Rightarrow D_n\}.$$

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
```

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.

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

Theorem: true
```

2.10.7 Backtracking expressions and deductions

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

$$\textbf{try}\ \{E_1\ \mid\ \ldots\ \mid\ E_n\} \tag{2.14}$$

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, that is, 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 (2.14), except that the alternatives are deductions rather than expressions:

$$\textbf{try}\ \{D_1\ \mid\ \ldots\ \mid\ D_n\}$$

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]

```
> 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

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

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

$$\texttt{lambda (n) (n times n).}$$

It is also 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 pass a procedure as an argument to another procedure. For example, the built-in `map` procedure takes a unary 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 $L = \texttt{[}V_1 \cdots V_n\texttt{]}$, then

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

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

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)

> (!commute-and B C)

Theorem: (and C B)

> (!commute-and A B)

standard input:3:15: Error: Failed application of left-and---the sentence
(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

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

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:

```
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

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

we can write

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

or, in prefix,

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

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** phrase 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

```
match F {
  π1  => D1
| π2  => D2
    ⋮
| πn  => Dn
}
```

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.

As we already mentioned in Section 2.10.6, to evaluate a **match** proof of the above form in an assumption base β and an environment ρ,[25] we start by evaluating the discriminant F in β and ρ, to obtain a value V_F.[26] We then start comparing the value V_F against the patterns π_i sequentially, $i = 1, \ldots, n$, to determine if it matches any of them. When we encounter the first pattern π_j that is successfully matched by V_F under some set of bindings $\rho' = \{I_1 \mapsto V_1, \ldots, I_k \mapsto V_k\}$, we evaluate the corresponding deduction D_j in the context of β and ρ *augmented with* ρ';[27] the result of that evaluation becomes the result of the entire **match** deduction.

In general, an attempt to match a value V against a pattern π will either result in failure, indicating that π does not reflect the structure of V; or else it will produce a matching environment $\rho' = \{I_1 \mapsto V_1, \ldots, I_k \mapsto V_k\}$, signifying that V successfully matches π under the bindings $I_j \mapsto V_j, j = 1, \ldots, k$. In the latter case we say that V *matches* π *under* ρ'.

The underscore _ is the wildcard pattern that is matched by any value whatsoever.

Suppose, for example, that the discriminant is the sentence `(true | ~ false)`. This sentence:

- matches the pattern `(p1 | p2)` under {`p1` ↦ `true`, `p2` ↦ `(not false)`};
- matches the pattern `p` under {`p` ↦ `(or true (not false))`};
- matches `(or true (not q))` under {`q` ↦ `false`};
- matches all three patterns _, `(or _ _)`, and `(or _ (not _))` under the empty environment {}.

Here are the first three examples in Athena:

```
> define discriminant := (true | ~ false)

Sentence discriminant defined.

> match discriminant {
    (p1 | p2) => (print "Successful match with p1: " p1 "\nand p2: " p2)
  }

Successful match with p1: true
and p2:
(not false)
```

25 Recall that an environment is a finite function mapping identifiers to values. Athena maintains a global environment that holds all the definitions made by the user (as well as built-in definitions). For instance, when a user issues a directive like **define** `p := (true | false)`, the global environment is extended with the binding `p` ↦ `(or true false)`. Since it is a finite function, an environment can be viewed as a finite set of identifier-value bindings, where each binding is an ordered pair consisting of an identifier I and a value V. We typically write such a binding as $I \mapsto V$.

26 We ignore the store and symbol set here because they do not play a central role in what we are discussing.

27 The result of augmenting (or "extending") an environment ρ_1 with another environment ρ_2 is the unique environment that maps an identifier I to $\rho_2(I)$, if I is in the domain of ρ_2; or to $\rho_1(I)$ otherwise.

```
Unit: ()

> match discriminant {
    p => (print "Successful match with p: " p)
  }

Successful match with p:
(or true
    (not false))

Unit: ()

> match discriminant {
    (or true (not q)) => (print "Successful match with q: " q)
  }

Successful match with q: false

Unit: ()
```

Patterns are generally written in prefix, but binary sentential constructors (as well as function symbols) can also appear inside patterns in infix, as seen in the first example above. Thus, for instance, `(and p1 (not (or p2 p3)))` and

```
(p1 & (~ (p2 | p3)))
```

are two equivalent patterns. However, patterns must always be fully parenthesized, so we cannot omit parentheses and rely on precedence and associativity conventions to determine the right grouping. For example, a pattern such as `(p & q ==> r)` will not have the intended effect; the pattern should be written as `((p & q) ==> r)` instead.

Sentences are not the only values on which we can perform pattern matching. We can also pattern-match terms, lists, and any combination of these. Consider, for instance, the following patterns:

1. `t`
2. `(mother t)`
3. `(mother (father person))`
4. `(father _)`

The term `(mother (father ann))` matches the first pattern under

$$\{\texttt{t} \mapsto \texttt{(mother (father ann))}\};$$

it matches the second pattern under $\{\texttt{t} \mapsto \texttt{(father ann)}\}$; it matches the third pattern under $\{\texttt{person} \mapsto \texttt{ann}\}$; and it does not match the fourth pattern.

```
define discriminant := (mother father ann)

> match discriminant {
    t => (print "Matched with t: " t)
  }

Matched with t:
(mother (father ann))

Unit: ()

> match discriminant {
    (mother t) => (print "Matched with t: " t)
  }

Matched with t:
(father ann)

Unit: ()

> match discriminant {
    (mother (father t)) => (print "Matched with t: " t)
  }

Matched with t:  ann

Unit: ()

> match discriminant {
    (father _) => (print "Matched...")
  }

standard input:1:1: Error: match failed---the term
(mother (father ann))
did not match any of the given patterns.
```

Term variables such as `?x:Boolean` are themselves Athena values, and hence can become bound to pattern variables. For example, the term `(union ?s1 ?s2)` matches the pattern `(union left right)` under the bindings $\{$`left` $\mapsto$ `?s1:Set`, `right` $\mapsto$ `?s2:Set`$\}$. Any occurrence of a term variable inside a pattern acts as a constant—it can only be matched by that particular variable. Hence, the pattern `(S ?n)` can only be matched by one value: the term `(S ?n)`.

Quantified sentences can also be straightforwardly decomposed with patterns. Consider, for instance, the pattern `(forall x p)`. The sentence

```
(forall ?human . male father ?human)
```

will match this pattern under the bindings

$\{$`x` $\mapsto$ `?human:Person`, `p` $\mapsto$ `(male (father ?human:Person))`$\}$.

The sentence

```
(forall ?x . exists ?y . ?x subset ?y & ?x =/= ?y)
```

will also match, under

$$\{\texttt{x} \mapsto \texttt{?x:Set}, \texttt{p} \mapsto \texttt{(exists ?y:Set . ?x subset ?y \& ?x =/= ?y)}\}.$$

Any identifier inside a pattern that is not a function symbol or a sentential constructor or quantifier (such as `if`, `forall`, etc.) is interpreted as a pattern variable, and can become bound to a value during pattern matching. For instance, assuming that `joe` has *not* been declared as a function symbol, the term `(father ann)` will match the pattern `(father joe)` under $\{\texttt{joe} \mapsto \texttt{ann}\}$. But if `joe` has been introduced as a function symbol, then it can no longer serve as a pattern variable, that is, it cannot become bound to any values. It can still appear inside patterns, but it can only match itself—the function symbol `joe`. Thus, the only value that will match the pattern `(mother joe)` at that point (after `joe` has been declared as a function symbol) is the term `(mother joe)` itself, and nothing else. So it is crucial to distinguish function symbols (and of course sentential constructors and quantifiers) from regular identifiers inside patterns.

Athena also supports nonlinear patterns, where multiple occurrences of the same pattern variable are constrained to refer to the same value. Consider, for instance, the pattern `(p | p)`. The sentence `(true | true)` matches this pattern under $\{\texttt{p} \mapsto \texttt{true}\}$; but the sentence `(true | false)` does not. Likewise, the terms `(union null null)` and

```
(union (intersection ?x ?y) (intersection ?x ?y))
```

match the pattern `(union s s)`, but the term `(union ?foo null)` does not.

Lists are usually taken apart with two types of patterns: patterns of the form

$$\texttt{(}\textbf{\texttt{list-of}}\ \pi_1\ \pi_2\texttt{)}$$

and those of the form $\texttt{[}\pi_1 \cdots \pi_n\texttt{]}$. The first type of pattern matches any nonempty list

$$\texttt{[}V_1 \cdots V_k\texttt{]}$$

with $k \geq 1$ and such that V_1 matches π_1 and the tail $\texttt{[}V_2 \cdots V_k\texttt{]}$ matches π_2. For example, the three-element list

```
[zero ann (father peter)]
```

matches the pattern `(list-of head tail)` under the bindings

$$\{\texttt{head} \mapsto \texttt{zero}, \texttt{tail} \mapsto \texttt{[ann (father peter)]}\}.$$

The second kind of pattern, $\texttt{[}\pi_1 \cdots \pi_n\texttt{]}$, matches all and only those n-element lists $\texttt{[}V_1 \cdots V_n\texttt{]}$ such that V_i matches π_i, $i = 1, \ldots, n$. For instance, `[ann peter]` matches the

pattern `[s t]` under {`s` ↦ `ann`, `t` ↦ `peter`}; but it does not match the pattern `[s s]`—a nonlinear pattern that is only matched by two-element lists with identical first and second elements.

These types of patterns can be recursively combined in arbitrarily complicated ways. For instance, the pattern

```
[((p & q) ==> (~ r)) (list-of (forall x r) _)]
```

is matched by any two-element list whose first element is a conditional of the form

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

and whose second element is any nonempty list whose first element is a universal quantification whose body is the sentence `r` that appears as the body of the negation in the consequent of the aforementioned conditional.

An arbitrary sentence of the form

$$(\circ\ p_1 \cdots p_n)$$

with $\circ \in$ {`not`, `and`, `or`, `if`, `iff`} can match a pattern of the form

$$((\texttt{some-sent-con}\ I)\ \pi_1 \cdots \pi_n),$$

provided that each p_i matches π_i in turn, $i = 1, \ldots, n$, in which case I will be bound to $\circ$:

```
> match (A & ~ B) {
    ((some-sent-con sc) left right) => [sc left right]
  }

List: [and A (not B)]

> match (A ==> B) {
    ((some-sent-con sc) left right) => [sc left right]
  }

List: [if A B]
```

A sentence of the form $(\circ\ p_1 \cdots p_n)$ can also match a pattern of the form

$$((\texttt{some-sent-con}\ I)\ \pi)$$

when π is a list pattern. Here I will be bound to $\circ$ and the list $[p_1 \cdots p_n]$ will be matched against π. For example, the following procedure takes any sentence p, and provided that p is an application of a sentential constructor sc to some subsentences $p_1 \cdots p_n$, it returns a pair consisting of sc and the sentences $p_1 \cdots p_n$ listed in reverse order:

```
> define (break-sentence p) :=
    match p {
      ((some-sent-con sc) (some-list args)) => [sc (rev args)]
    }
```

```
Procedure break-sentence defined.

> (break-sentence (~ true))

List: [not [true]]

> (break-sentence (and A B C))

List: [and [C B A]]

> (break-sentence (A | B))

List: [or [B A]]

> (break-sentence (false ==> true))

List: [if [true false]]

> (break-sentence (iff true true))

List: [iff [true true]]

> (break-sentence true)

Error, standard input, 2.5: match failed---the term true
did not match any of the given patterns.
```

Another useful type of pattern is the **where** pattern, of the form

$$(\pi \ \textbf{where} \ E),$$

where E is an expression that may contain pattern variables from π. The idea here is that we first match a discriminant value V against π, and if the match succeeds with a set of bindings, then we proceed to evaluate E in the context of those bindings (on top of the environment in which V was obtained). The overall pattern succeeds (with that same set of bindings) if the evaluation of E produces `true` and fails otherwise. For instance, the following matches a list whose head is an even integer:

```
define (first-even? L) :=
  match L {
    ((list-of x _) where (even? x)) => true
  | _ => false
  }
```

We have only scratched the surface of Athena's pattern-matching capabilities here. The subject is covered in greater detail in Section A.4.

2.12 Directives

In addition to expressions and deductions, the user can give various *directives* as input to Athena. These are commands that direct Athena to do something, typically to enter new information or adjust some setting about how it processes its input or how it displays its output. Most such directives have been mentioned already: **load** (page 22), **set-precedence** (page 32), **left-assoc**/**right-assoc** (page 33), **define** (page 40), **assert** (page 43), **clear-assumption-base**/**retract** (page 44), and **set-flag** `auto-assert-dt-axioms` (page 48). In this section we describe a couple of **set-flag** directives for controlling output, and the next section continues with the **overload** directive for adapting function symbols to have different meanings depending on context.

- **set-flag** `print-var-sorts` *s*, where *s* is either the string `"on"` or the string `"off"`. The default value is `"on"`. When set to `"off"`, Athena will not print out the sorts of variables. Examples:

```
> set-flag print-var-sorts "off"

OK.

> (?x = ?y)

Term: (= ?x ?y)

> set-flag print-var-sorts "on"

OK.

> (?x = ?y)

Term: (= ?x:'T185 ?y:'T185)
```

- **set-flag** `print-qvar-sorts` *s*, where *s* is again either the string `"on"` or `"off"`. When `print-vars-sorts` is turned off, variable sorts in the body of a quantified sentence are not printed, but variable occurrences that immediately follow a quantifier occurrence continue to have their sorts printed. The printing of even those sorts can be disabled by turning off the flag `print-qvar-sorts`:

```
> set-flag print-qvar-sorts "off"

OK.

> (forall ?x . exists ?y . ?y > ?x)

Sentence: (forall ?x
```

```
            (exists ?y
              (> ?y ?x)))
```

Turning off variable sort printing can sometimes simplify output significantly, especially when there are several long polymorphic sorts involved, but keeping it on can often provide useful information.

2.13 Overloading

Say we have introduced `Plus` as a binary function symbol intended to represent addition on the natural numbers:

```
declare Plus: [N N] -> N
```

While we could go ahead and use `Plus` in all of our subsequent proofs, it might be preferable, for enhanced readability, if we could use + instead of `Plus`, since + is traditionally understood to designate addition. However, + is already used in Athena to represent addition on real numbers (i.e., it is already declared at the top level as a binary function symbol that takes two real numbers and produces a real number). This means that we cannot redeclare + to take natural numbers instead:

```
> declare +: [N N] -> N

Warning, standard input:1:9: Duplicate symbol---the name + is
already used as a function symbol.
```

We could, of course, simply *define* + to be `Plus`, but then we would lose the original meaning of + as a binary function symbol on the real numbers.

At the top level we can get around these difficulties by *overloading* + so that it can stand for `Plus` whenever that makes sense but revert to its original meaning at all other times:

```
> overload + Plus

OK.
```

We can now use + for both purposes: as an alias for `Plus`, to denote addition on natural numbers; and also to denote the original function, addition on real numbers. Which of these alternatives is chosen depends on the context. More specifically, it depends on the sorts of the terms that we supply as arguments to +. If the arguments to + are natural numbers, then + is understood as `Plus`. If, on the other hand, the arguments are not natural numbers, then

Athena infers that + is being used in its original capacity, to represent a function on real numbers:

```
1 > (?a + zero)
2
3 Term: (Plus ?a:N zero)
4
5 > (?a + 2.5)
6
7 Term: (a:Real + 2.5)
```

Here, on line 1, Athena interpreted + as `Plus`, because even though the variable `?a` was not annotated, the second argument was `zero`, a natural number. Hence, the only viable alternative was to treat + as `Plus`. On line 5, by contrast, + could not possibly be understood as `Plus`, since the second argument was `2.5`, of sort `Real`, and hence + was treated as it would have been normally treated prior to the overloading. If the sorts of the arguments to + are completely unconstrained, in which case both interpretations are plausible, then the most recently overloaded meaning takes precedence, in this case `Plus`:

```
> (?a + ?b)

Term: (Plus ?a:N ?b:N)
```

Essentially (if somewhat loosely), after a directive of the form **`overload`** f g has been issued, every time Athena encounters an application of the form $(f\ \cdots)$ it tries to interpret it as $(g\ \cdots)$. If that fails, then it interprets the application based on the original meaning of f.

This is not overloading in the conventional sense, because we do not directly declare + to have two distinct signatures, one of which expects two natural numbers as inputs and produces a natural number as output. Instead, we first introduce `Plus`, and then we essentially announce that + is to be used as an alias for `Plus` in any context in which it is sensible to do so. Thus, for instance, `(+ zero zero)` actually produces the term `(Plus zero zero)` as its output. So the name + here really is used simply as a synonym for `Plus`.

Note that after the overloading, + no longer denotes a function symbol. Rather, it denotes a binary procedure that does what was described above: It first tries to apply `Plus` to its two arguments, and if that fails, it tries to apply to them *whatever was previously denoted by* +, which in this case is a function symbol.[28] This process can be iterated indefinitely. For instance, after first overloading + to represent `Plus`, we might later overload it even further, say, to represent `::`, the reflexive constructor of the `List` datatype (see page 57):

28 Of course, + remains a function symbol in the current symbol set.

```
1  > overload + Plus
2
3  OK.
4
5  > (?a + ?b)
6
7  Term: (Plus ?a:N ?b:N)
8
9  > overload + ::
10
11 OK.
12
13 > (?a + ?b)
14
15 Term: (:: ?a:'T2
16           ?b:(List 'T2))
17
18 > (?a + zero)
19
20 Term: (Plus ?a:N zero)
21
22 > (?a + 3.14)
23
24 Term: (+ ?a:Real 3.14)
```

Here, after the second overloading on line 9, + denotes a binary procedure that takes two values v_1 and v_2 and tries to apply :: to them. If that fails, then it applies to v_1 and v_2 whatever was previously denoted by +, which in this case is the procedure that resulted from the first overloading, on line 1.

Multiple overloadings can be carried out in one fell swoop as follows:

```
> overload (+ Plus) (- Minus) (* Times) (/ Div)

OK.
```

The above is equivalent to the following sequence of individual directives:

```
overload + Plus
overload - Minus
overload * Times
overload / Div
```

The **overload** directive is most useful when we are working exclusively at the top level or inside a single module. Across different modules there is rarely a need for explicit overloading, since the same function symbol can be freely declared in multiple modules with different signatures. That is, two distinct modules M and N are allowed to declare a function symbol of one and the same name, f. No conflict arises because the enclosing modules serve to disambiguate the symbols: In one case we are dealing with $M.f$ and in the other

with $N.f$. So we could, for instance, develop our theory of natural numbers inside a module named, say, `N`, and then directly introduce a function symbol `+` inside `N` to designate addition, which would altogether avoid the introduction of `Plus`. This is, in fact, the preferred approach when developing specifications and proofs in the large. The alternative we have described here, **`overload`**, does not involve modules and can be used in smaller-scale projects (though it can also come in handy sometimes inside modules).

We have seen that if f is a function symbol, then after a directive like

`overload` f g

is issued, f will no longer denote the symbol in question; it will instead denote a procedure (recall that function symbols and procedures are distinct types of values). This raises the question of how we can now retrieve the *symbol* f. For instance, consider the first time we overload `+`:

```
> overload + Plus

OK

> +

Procedure: +
```

How can we now get hold of the actual function symbol `+`? (We might need the symbol itself for some purpose or other.) The newly defined procedure can still make terms with the symbol `+` at the top, so all we need to do is grab that symbol with the `root` procedure, though simple pattern matching would also work:

```
> (root (1 + 2))

Symbol: +
```

But probably the easiest way to obtain the function symbol after the corresponding name has been redefined via **`overload`** is to use the primitive procedure `string->symbol`, which takes a string and, assuming that the current symbol set contains a function symbol f of the same name as the given string, it returns f:

```
> (string->symbol "+")

Symbol: +
```

2.14 Programming

In this section we briefly survey some Athena features that are useful for programming.

2.14.1 Characters

A literal character constant starts with ‘ and is followed by either the character itself, if the character is printable, or by *d*, where *d* is a numeral of one, two, or three digits representing the ASCII code of the character:

```
> ‘A

Character: ‘A

> ‘\68

Character: ‘D
```

Standard escape and control sequences (also starting with ‘) are understood as well, for example, ‘`\n` indicates the newline character, ‘`\t` the tab, and so on.

2.14.2 Strings

A string is just a list of characters:

```
> (print (tail "hello world"))

ello world
Unit: ()
```

Built-in procedures like `rev` and `join` can therefore be directly applied to them.

2.14.3 Cells and vectors

A cell is a storage container that can hold an arbitrary value (possibly another cell). At a lower level of abstraction, it can be thought of as a constant pointer, pointing to a specific address where values can be stored. A cell containing a value *V* can be created with an expression of the form **`cell`** *V*. The contents of a cell *c* can be accessed by writing **`ref`** *c*. We can destructively modify the contents of a cell *c* by storing a value *V* in it (overwriting any previous contents) with an expression of the form **`set!`** *c V*.

```
> define c := cell 23

Cell c defined.

> ref c

Term: 23

> set! c "a string now..."
```

```
Unit: ()

> ref c

List: [‘a ‘\blank ‘s ‘t ‘r ‘i ‘n ‘g ‘\blank ‘n ‘o ‘w ‘. ‘. ‘.]
```

A vector is a fixed-length array of values. If N is an expression whose value is a nonnegative integer numeral and V is any value, then an expression of the form

make-vector N V

creates a new vector of length N, every element of which contains V. If A is a vector of size N and i is an index between 0 and $N-1$,

vector-sub A i

returns the value stored in the i^{th} element of A. To store a value V into the i^{th} element of A (assuming again that A is of size N and $0 \leq i < N$), we write

vector-set! A i V.

2.14.4 Tables and maps

A *table* is a dictionary ADT (abstract data type), implemented as a hash table mapping keys to values. Keys are hashable Athena values: characters, numbers, strings, but also terms, sentences, and lists of such values (including lists of lists of such values, and so on). Tables provide constant-time insertions and lookups, on average. The functionality of tables is accessible through the module `HashTable`,[29] which includes the following procedures:

1. `HashTable.table`: A nullary procedure that creates and returns a new hash table. Optionally, it can take an integer argument specifying the table's initial size.
2. `HashTable.add`: A binary procedure that takes a hash table T as its first argument and either a single key-value binding or a list of them as its second argument; and augments T with the given binding(s). Each binding is either a pair of the form `[`*key val*`]` or else a 3-element list of the form `[`*key* `-->` *val*`]`.[30] Multiple bindings are added from left to right. The unit value is returned.
3. `HashTable.lookup`: A binary procedure that takes a hash table T as its first argument and an arbitrary key as its second argument and returns the value associated with that key in T. It is an error if the key is not bound in T.

29 See Chapter 7 for more details on Athena's modules. For those familiar with `C++`, you can think of modules as namespaces similar to those of `C++`, with minor syntactic differences (e.g., `M.x` is used instead of `M::x`).

30 `-->` is a constant function symbol declared in Athena's library.

4. `HashTable.remove`: A binary procedure that takes a hash table T as its first argument and an arbitrary *key* as its second argument. It is an error if *key* is not bound in T. Otherwise, if *key* is bound to some *val*, then that binding is removed from T and *val* is returned.
5. `HashTable.clear`: A unary procedure that takes a hash table T and clears it (removes all bindings from it). The unit value is returned.
6. `HashTable.size`: A unary procedure that takes a hash table T and returns its size (the number of bindings in T).
7. `HashTable.table->string`: A unary procedure that takes a hash table and returns a string representation of it.
8. `HashTable.table->list`: A unary procedure that takes a hash table and returns a list of all the bindings in it (as a list of key-value pairs).

A *map* is also a dictionary ADT, but one that is implemented as a functional tree: Inserting a new key-value pair into a map m creates and returns another map m', obtained from m by incorporating the new key-value binding; the old map m is left unchanged. Maps provide logarithmic-time insertions and lookups. To apply a map m to a key k, we simply write `(`m k`)`. Thus, notationally, maps are applied just like procedures, although semantically they form a distinct type. The rest of the interface of maps is accessible through the module `Map`, which includes the procedures described below. Note that map keys can only come from types of values that admit of computable total orderings. These coincide with the hashable value types described above. Any value can be used as a map key, but if it's not of the right type (admitting of a computable total ordering) then its printed representation (as a string) will serve as the actual key.

- `Map.make`: A unary procedure that takes a list of key-value bindings

  $$\texttt{[[}key_1\ \ val_1\texttt{]}\ \cdots\ \texttt{[}key_n\ \ val_n\texttt{]]}$$

  and returns a new map that maps each key_i to val_i, $i = 1, \ldots, n$. The same map can also be created from scratch with the custom notation:

  $$|\{key_1 := val_1, \ldots, key_n := val_n\}|.$$

- `Map.add`: A binary procedure that takes a map m and a list of bindings (with each binding represented as a `[`*key* *val*`]` pair) and returns a new map m' that extends m with the given bindings, added from left to right.
- `Map.remove`: A binary procedure that takes a map m and a key k and returns the map obtained from m by removing the binding associated with k. If m has no such binding, it is returned unchanged.
- `Map.size`: A unary procedure that takes a map and returns the number of bindings in it.

- `Map.keys`: A unary procedure that takes a map and returns a list of all the keys in it.
- `Map.values`: Similar, except that it returns the list of values in the map.
- `Map.key-values`: Similar, but a list of `[`*key val*`]` pairs is returned instead.
- `Map.map-to-values`: A binary procedure that takes a map m and a unary procedure f and returns the map obtained from m by applying f to the values of m.
- `Map.map-to-key-values`: Similar, except that the unary procedure f expects a pair as an argument, and f is applied to each key-value pair in the map.
- `Map.apply-to-key-values`: A binary procedure that takes a map m and a side effect-producing unary procedure f and applies f to each key-value pair in m. The unit value is returned.
- `Map.foldl`: A ternary procedure that takes a map m, a ternary procedure f, and an initial value V, and returns the value obtained by left-folding f over all key-value pairs of m (passing the key as the first argument to f and the value as the second; the third argument of f serves as the accumulator), and using V as the starting value.

2.14.5 While loops

A **`while`** expression is of the form **`while`** E_1 E_2. The semantics of such loops are simple: As long as E_1 evaluates to `true`, E_2 is evaluated. Such loops are rarely used in practice. Even when side effects are needed, (tail) recursion is a better alternative.

2.14.6 Expression sequences

A sequence of one or more phrases ending in an expression can be put together with an expression of this form:

$$\texttt{(}\textbf{\texttt{seq}}\ F_1 \cdots F_n\ E\texttt{)}.$$

The phrases $F_1, \ldots, F_n, E$ are evaluated sequentially and the value of E is finally returned as the value of the entire **`seq`** expression. This form is typically used for code with side effects.

2.14.7 Recursion

Phrases of the form

$$\textbf{\texttt{letrec}}\ \{\, I_1 := E_1 \,;\ \cdots\ ;\ I_n := E_n \,\}\ F$$

allow for mutually recursive bindings: An identifier I_j might occur free in any E_i, even for $i \leq j$. The precise meaning of these expressions is given by a desugaring in terms of **`let`** and cells (via **`set!`** expressions), but an intuitive understanding will suffice here. In most

cases the various E_i are **lambda** or **method** expressions. The phrase F is the *body* of the **letrec**. If F is an expression then the entire **letrec** is an expression, otherwise the **letrec** is a deduction. For example, the following code defines two mutually recursive procedures for determining the parity of an integer:[31]

```
define [even? odd?] :=
  letrec {E := lambda (n)
                 check {(n equal? 0) => true
                       | else => (O n minus 1)};
          O := lambda (n)
                 check {(n equal? 0) => false
                       | else => (E n minus 1)}}
    [lambda (n) (E (abs n))
     lambda (n) (O (abs n))]
```

where abs is a primitive procedure that returns the absolute value of a number (integer or real).

Explicit use of **letrec** is not required when defining procedures or methods at the top level. The default notation for defining procedures, for instance, essentially wraps the body of the definition into a **letrec**. So, for example, we can define a procedure to compute factorials as follows:

```
define (fact n) :=
  check {(n less? 2) => 1
       | else => (n times fact n minus 1) }
```

However, if we want to define an inner recursive procedure (introduced inside the body of a procedure defined at the top level), then we need to use **letrec**:

```
define (f ···) :=
   ···
   letrec {g := lambda (···)
                  ··· (g ···) ···}
     ···
```

2.14.8 Substitutions

Formally, a *substitution* θ is a finite function from variables to terms:

$$\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}, \tag{2.15}$$

where each term t_i is of the same sort as the variable x_i. We say that the set of variables $\{x_1, \ldots, x_n\}$ constitutes the *support* of θ. In Athena, substitutions form a distinct type of

31 This is purely for illustration purposes; a much better way to determine parity is to examine the remainder after division by 2 (using the mod procedure).

values. A substitution of the form (2.15) can be built with the unary procedure `make-sub`, by passing it as an argument the list of pairs `[[`x_1 t_1`]` $\cdots$ `[`x_n t_n`]]`. For example:

```
> (make-sub [[?n zero] [?m (S ?k)]])

Substitution:
{?n:N --> zero
?m:N --> (S ?k:N)}
```

Note the format that Athena uses to print out a substitution: {x_1 `-->` t_1 $\cdots$ x_n `-->` t_n}. The support of a substitution can include variables of different sorts, say:

```
> (make-sub [[?counter (S zero)] [?flag true]])

Substitution:
{?counter:N --> (S zero)
?flag:Boolean --> true}
```

It may also include polymorphic variables:

```
> (make-sub [[?list (:: ?head ?tail)]])

Substitution: {?list:(List 'S) --> (:: ?head:'S ?tail:(List 'S))}
```

The support of a substitution can be obtained by the unary procedure `supp`. The empty substitution is denoted by `empty-sub`. A substitution θ of the form (2.15) can be *extended* to incorporate an additional binding $x_{n+1} \mapsto t_{n+1}$ by invoking the ternary procedure `extend-sub` as follows:

$$\texttt{(extend-sub } \theta \ x_{n+1} \ t_{n+1}\texttt{)}.$$

If we call `(extend-sub` θ x t`)` with a variable x that already happens to be in the support of θ, then the new binding for x will take precedence over the old one (i.e., the resulting substitution will map x to t).

Substitutions can be *applied* to terms and sentences. In the simplest case, the result of applying a substitution θ of the form (2.15) to a term t is the term obtained from t by replacing every occurrence of x_i by t_i. The syntax for such applications is the same syntax used for procedure applications: `(`θ t`)`.[32] For example:

```
define theta := (make-sub [[?counter (S zero)] [?flag true]])

> theta

Substitution:
{?counter:N --> (S zero)
?flag:Boolean --> true}
```

32 When we use conventional mathematical notation, we might write such an application as $\theta(t)$ instead.

```
> (theta ?flag)

Term: true

> (theta (?foo = S ?counter))

Term: (= ?foo:N
         (S (S zero)))
```

The result of applying a substitution θ of the form (2.15) to a sentence p, denoted by `(`θ p`)`, is the sentence obtained from p by replacing every free occurrence of x_i by t_i.

In many applications, substitutions are obtained incrementally. For instance, first we may obtain a substitution such as θ_1 `= {?a:N ↦ (S ?b)}`, and then later we may obtain another one, for example,

$$\theta_2 \texttt{ = \{?b:N} \mapsto \texttt{ zero\}}.$$

We want to combine these two into a single substitution that captures the information provided by both. We can do that with an operation known as substitution *composition*. In this case, the composition of θ_2 with θ_1 yields the result:

$$\theta_3 \texttt{ = \{?a:N} \mapsto \texttt{ (S zero), ?b:N} \mapsto \texttt{ zero\}}.$$

We can think of θ_3 as combining the information of θ_1 and θ_2. To obtain the composition of θ_2 with θ_1, we apply the binary procedure `compose-subs` to the two substitutions. In general, for any

$$\theta_1 = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

and

$$\theta_2 = \{y_1 \mapsto s_1, \ldots, y_m \mapsto s_m\}$$

we have:

$$\texttt{(compose-subs } \theta_2 \ \theta_1\texttt{)} = \{x_1 \mapsto (\theta_2\ t_1), \ldots, x_n \mapsto (\theta_2\ t_n), y_{i_1} \mapsto s_{i_1}, \ldots, y_{i_k} \mapsto s_{i_k}\},$$

where the set $\{y_{i_1}, \ldots, y_{i_k}\} \subseteq \{y_1, \ldots, y_m\}$ contains every variable in the support of θ_2 that is "new" as far as θ_1 is concerned, that is, every y_i whose name is not the name of any x_j.

Two very useful operations on terms that return substitutions are *term matching* and *unification*. We say that a term s *matches* a term t if and only if we can obtain s from t by consistently replacing variables in t by certain terms. Thus, the term t is viewed as a template or a pattern. The variables of t act as placeholders—empty boxes to be filled by sort-respecting terms. Plugging appropriate terms into these placeholders produces the term s. We say that s is an *instance* of t.

For example, `s = (zero Plus S ?a)` matches the term `t = (?x Plus ?y)`, and hence s is an instance of t, because we can obtain s from t by substituting `zero` for `?x` and `(S ?a)`

for ?y. A more precise way of capturing this relation is to say that s matches t iff there exists a substitution that produces s when applied to t. The binary procedure match-terms efficiently determines whether a term (the first argument) matches another (the second). If it does not, false is returned, otherwise a matching substitution is produced. Matching is a fundamental operation of central importance that arises in many areas of computer science, from programming languages and databases to artificial intelligence. In this book, matching will be widely used in dealing with equational proofs by means of *rewriting*. For example:

```
> (match-terms (null union ?x) (?s1 union ?s2))

Substitution:
{?s2:Set --> ?x:Set
?s1:Set --> null}

> (match-terms (father joe) (mother ?x))

Term: false
```

There is a corresponding primitive procedure match-sentences that extends this notion to sentences. Roughly, a sentence p matches a sentence q iff either both p and q are atomic sentences and p matches q just as a term matches another term; or else both are complex sentences built by the same sentential constructor or quantifier and their corresponding immediate subsentences match recursively. In particular, when p and q are quantified sentences $(Q\ x\ p')$ and $(Q\ y\ q')$, respectively (where Q is either the universal or the existential quantifier), we proceed by recursively matching $\{x \mapsto v\}(p')$[33] against $\{y \mapsto v\}(q')$, where v is some fresh variable of the same sort as x and y, with the added proviso that v cannot appear in the resulting substitution (to ensure that if any variables appear in a resulting substitution, they are among the free variables of the two sentences that were matched). Some examples:

```
> (match-sentences (~ joe siblings ann) (~ ?x siblings ?y))

Substitution:
{?y:Person --> ann
 ?x:Person --> joe}

> (match-sentences (forall ?x . ?x siblings joe)
                   (forall ?y . ?y siblings ?w))

Substitution: {?w:Person --> joe}
```

33 Recall that for any sentence p, variable v, and term t, we write $\{v \mapsto t\}(p)$ for the sentence obtained from p by replacing every free occurrence of v by t, taking care to rename bound variables as necessary to avoid variable capture.

We conclude with a brief discussion of unification. Informally, to unify two terms s and t is to find a substitution that renders them identical, that is, a substitution θ such that (`θ s`) and (`θ t`) are one and the same term. Such a substitution is called a *unifier* of s and t. Two terms are *unifiable* if and only if there exists a unifier for them. Consider, for instance, the two terms $s =$ `(S zero)` and $t =$ `(S ?x)`. They are unifiable, and in this case the unifier is unique: {`?x` ↦ `zero`}. Unifiers need not be unique. For instance, the terms `(S ?x)` and `?y:N` are unifiable under infinitely many substitutions.

The binary procedure `unify` can be used to unify two terms (i.e., to produce a unifier for them); `false` is returned if the terms cannot be unified. Note that unifiability is not the same as matching. Neither of two terms might match the other, but the two might be unifiable nevertheless. Athena's procedures for matching and unification handle polymorphic inputs as well, consistent with the intuitive understanding of a polymorphic term (or sentence) as a collection of monomorphic terms (or sentences).

2.15 A consequence of static scoping

Athena is statically scoped. Roughly speaking, this means that free identifier occurrences inside the body of a procedure (or method) get their values from the lexical environment in which the procedure (or method) was defined. It also means that consistent renaming of bound identifier occurrences does not affect the meaning of a phrase. This is in contrast to *dynamic scoping*, in which free identifier occurrences inside the body of a procedure (or method) get their values from the lexical environment in which the procedure (or method) is called, not the environment in which it was defined. Almost all modern programming languages use static scoping, because dynamic scoping is conducive to subtle errors that are difficult to recognize and debug. For our purposes in this textbook we need not get into all the complexities of static vs. dynamic scoping, but it is worth noting a consequence that could seem puzzling to the novice. First, consider the following definitions of procedures `f` and `g`:

```
1  > define (f x) := (x plus x)
2
3  Procedure f defined.
4
5  > define (g x) := ((f x) plus 3)
6
7  Procedure g defined.
8
9  > (g 5)
10
11 Term: 13
```

Suppose we now realize we should have defined `f` as squaring rather than doubling `x`, so we redefine it:

```
> define (f x) := (x times x)

Procedure f defined.

> (g 5)

Term: 13
```

Why do we still get the same result? Because in the earlier code, at the point where we defined `g` (line 5), the free occurrence of `f` in the body of `g` referred to the doubling procedure defined in line 1. This is a statically fixed binding, unchanged when we bind `f` to a new procedure in the second definition. Thus, we must also redefine `g` in order to have a definition that binds `f` to the new function, by reentering the (textually) same definition as before:

```
> define (g x) := ((f x) plus 3)

Procedure g defined.

> (g 5)

Term: 28
```

In general, if you are interactively entering a series of definitions and you then revise one or more of them, you'll also need to update the definitions of other values that refer to the ones you have redefined. Of course, if you enter all the definitions in a file that you then load, things are simpler: You can just go back and edit the text of the definitions that need changing and then reload the file.

2.16 Miscellanea

Here we describe some useful features of Athena that do not neatly fall under any of the subjects discussed in the preceding sections.

1. *Short-circuit Boolean operations*: `&&` and `||` perform the logical operations AND and OR on the two-element set {`true`, `false`}. They are special forms rather than primitive procedures precisely in order to allow for short-circuit evaluation.[34] In particular, to evaluate (`&&` $F_1 \cdots F_n$), we first evaluate F_1 to get a value V_1. V_1 must be either `true` or `false`, otherwise an error occurs. If it is `false`, the result is `false`; if it is `true`, then we proceed to evaluate F_2, to get a value V_2. Again, an error occurs if V_2 is neither `true` nor `false`. Assuming no errors, we return `false` if V_2 is `false`; otherwise V_2 is `true`, so we

34 Because Athena is a strict (call-by-value) language, if, say, `&&` were just an ordinary procedure, then all of its arguments would have to be fully evaluated before the operation could be carried out, and likewise for `||`.

proceed with F_3, and so on. If every F_i is `true` then we finally return `true` as the result. We evaluate `(||` $F_1 \cdots F_n$`)` similarly, but with the roles of `true` and `false` reversed.

2. *Fresh variables*: There is a predefined procedure `fresh-var` that will return a *fresh variable*: a variable whose name is guaranteed to be different from that of every other variable previously encountered in the current Athena session. When called with zero arguments, the procedure returns a fresh variable whose sort is completely unconstrained:

```
> (fresh-var)

Term: ?v1:'T190
```

A fresh variable of a specific sort can be created by passing the desired sort as a string argument to `fresh-var`:

```
> (fresh-var "Int")

Term: ?v2:Int

> (fresh-var "(Pair 'T Boolean)")

Term: ?v3:(Pair 'T192 Boolean)

> ?v4

Term: ?v4:'T193

> (fresh-var)

Term: ?v5:'T194
```

If we want the name of the fresh variable to start with a prefix of our choosing rather than the default `v`, we can pass that prefix as a second argument, in the form of a meta-identifier:

```
> (fresh-var "Int" 'foo)

Term: ?foo253:Int
```

The ability to generate fresh variables is particularly useful when implementing theorem provers.

3. *Dynamic term construction*: Sometimes we have a function symbol f and a list of terms `[`$t_1 \cdots t_n$`]`, and we want to form the term `(`f $t_1 \cdots t_n$`)`, but we cannot apply f directly because the number n is not statically known (indeed, often both f and the list of terms are input parameters). For situations like that there is the binary procedure `make-term`, which takes a function symbol f and a list of terms `[`$t_1 \cdots t_n$`]` and returns `(`f $t_1 \cdots t_n$`)`, provided that this term is well sorted:

```
> (make-term siblings [joe ann])

Term: (siblings joe ann)
```

4. *Free variable computation*: The primitive unary procedure `free-vars` (also defined as `fv`) takes any sentence p and returns a list of those variables that have free occurrences in p:

```
define p := (?x < ?y + 1 & forall ?x . exists ?z . ?x = ?z)

> (fv p)

List: [?x:Int ?y:Int]

> (fv (forall ?x . ?x = ?x))

List: []
```

5. *Free variable replacement*: The operation of safely replacing every free occurrence of a variable v inside a sentence p by some term t, denoted by $\{v \mapsto t\}(p)$ (see page 40), is carried out by the primitive ternary procedure `replace-var`. Specifically,

   `(replace-var` v t p`)`

   produces the sentence obtained from p by replacing every free occurrence of v by t, renaming as necessary to avoid variable capture.

6. *Dummy variables*: Sometimes we need a "dummy" variable whose name is unimportant. We can use the underscore character to generate a fresh dummy variable, that is, a variable that has not yet been seen during the current session:

```
> _

Term: ?v70:'T646

> _

Term: ?v71:'T647
```

   Observe that we get a different variable each time, first `?v70` and then `?v71`. That is what makes these variables fresh. The sorts of these variables are completely unconstrained, which makes the variables maximally flexible: They can appear in any context whatsoever and take on the locally required sorts.

```
> (father _)

Term: (father ?v73:Person)
```

```
> (_ in _)

Term: (in ?v350:'T4428
          ?v351:(Set 'T4428))
```

7. *Proof errors*: A primitive nullary method `fail` halts execution when applied and raises an error:

```
> (!fail)

standard input:1:1: Error: Proof failure.
```

A related method, `proof-error`, has similar behavior except that it takes a string as an argument (an error message of some kind), and prints that string in addition to raising an error.

8. *Patterns inside* **`let`** *phrases*: The syntax of **`let`** phrases is somewhat more flexible than indicated in (2.11). Specifically, instead of identifiers I_j, we can have patterns appearing to the left of the assignment operators `:=`. These can be term patterns, sentential patterns, list patterns, or a mixture thereof. For example:

```
> let {[x y] := [1 2]}
    [y x]

List: [2 1]

> let {[(siblings L (father R))] := [(siblings joe (father ann))]}
    [L R]

List: [joe ann]
```

9. *last-val*: At the beginning of each iteration of the read-eval-print loop, the identifier `last-val` denotes the value of the most recent phrase that was evaluated at the top level:

```
> (rev [1 2])

List: [2 1]

> last-val

List: [2 1]
```

10. *Primitive methods*: A primitive method in this context refers to an explicitly defined method M whose body is an *expression* E (rather than a deduction, as is the usual requirement for every nonprimitive 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 $I_1 \cdots I_n$ refer to the arguments of M. One can think of Athena's primitive methods—such as `mp`—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 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 term, of the form $?I:S$, where S is a sort. They act as syntactic placeholders. We use the letters x, y, z, and v to range over variables.
  2. *Sentences*, such as `(= 1 1)` or `(not false)`. 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.[36]
  8. *Methods*, whether they are primitive (such as `both`) or user-defined.

---

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 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."

9. ASCII *characters*.
10. *Substitutions*, which are finite maps from term variables to terms.
11. *Cells* and *vectors*, which can be used to store and 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]

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, 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 `[(`$f\ V_1$`)` $\cdots$ `(`$f\ V_n$`)]`.
- `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 or equal to 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 singleton 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. □