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 anonynous) 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 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*, *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:

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

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

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 practical advantages.

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

an outright contradiction, without any restrictions whatsoever. But a deduction has to play 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 restricted.

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, what the value residing there is. Finally, we need a *symbol set*, which is a finite collection of function symbols and their signatures (we will explain what these are shortly); this information is needed primarily in order to properly interpret patterns.

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:

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

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's programming language. 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 Athena 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 Athena 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 s-expression 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

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

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 term 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 simply "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]

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 help to prescribe the meaning of a symbol.

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 the factorial function, 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)

Term: 4
```

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

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.

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; 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.

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 (father joe))
```

is a legal term. Its root is the symbol `father`, and it has only one child, the term `(father 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

> (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.

8 A more common syntax for such terms that you may be familiar with 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.

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*, e.g., `(father true)`. An attempt to construct this term would result in a "sort error" because the function symbol `father` requires an argument from the `Person` domain and we are giving it a `Boolean` instead. Athena performs sort checking automatically and will detect and report any ill-sorted terms as errors:

```
> (father true)

standard input:1:2: Error: Unable to infer a sort for the term:(father true)

(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 also gives a parenthetical indication (line 5 above) of why it was unable to infer a sort for the term.

Note that we speak of "ill-sorted" terms, "sort errors," etc., rather than ill-typed terms, type errors, etc. 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, etc. 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 the 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 (father joe))
```

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

```
(father joe)
(father father joe)
(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 2.16.

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 inpect the precedence of all 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 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

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.

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.

The combination of negation with equality has its own shorthand:

Connective	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 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:

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
```

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 between the respective propositional contents, relationships that cannot be captured by any truth-functional treatment of sentential constructors. For the purposes of this book, however, we will ignore such debates and will freely “speak with the vulgar” to say that `(`p `==>` q`)` means that p implies q.

```
            (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.

Note that 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 ?S2 ?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`:

```
1  > (?x = joe & male ?x)
2
3  Sentence: (and (= ?x:Person joe)
4                 (male ?x:Person))
5
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 will result in

$\texttt{(exists ?x:Real . ?x:Real > 2 * }\boxed{\texttt{?x:Real}}\texttt{)}$,

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' = `(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 *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))
```