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 ahold 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 either by the character itself, if the character is printable, or else 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 **`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 essentially an 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 $0 \le i < N$), 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, etc.). Tables provide constant-time insertion 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 now you can think of modules as namespaces very similar to those of `C++`, with trivial syntactic differences (e.g., `M.x` is used instead of `M::x`).

30 `-->` is a constant function symbol 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 $T$ and returns a string representation of $T$.
8. `HashTable.table->list`: A unary procedure that takes a hash table $T$ and returns a list of all the bindings in $T$ (as a list of 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 of value. 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

  $$[[key_1 \ \ val_1] \ \cdots \ [key_n \ \ val_n]]$$

  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 being 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 $m$ 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

$$\texttt{while}\ E_1\ E_2.$$

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:

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

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

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

allow for mutually recursive bindings: An identifier $I_j$ might occur free in any $E_i$, even for $i \le j$. 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* $\theta$ is a finite function from variables to terms:

$$\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}, \tag{2.20}$$

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 $\theta$. 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 in Athena).

values. A substitution of the form (2.20) 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, e.g.:

```
> (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 $\theta$ of the form (2.20) 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` $\theta$ $x$ $t$`)` with a variable $x$ that already happens to be in the support of $\theta$, 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 $\theta$ of the form (2.20) 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: `(`$\theta$ $t$`)`.[32] For example:

```
> theta

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

> (theta ?flag)
```

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

```
Term: true

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

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

The result of applying a substitution $\theta$ of the form (2.20) to a sentence $p$, denoted by $(\theta\ \ p)$, 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 $\theta_1$ = {?a:N $\mapsto$ (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 $\theta_2$ with $\theta_1$ yields the result:

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

We can think of $\theta_3$ as combining the information of $\theta_1$ and $\theta_2$. To obtain the composition of $\theta_2$ with $\theta_1$ in Athena, 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 $\theta_2$ that is "new" as far as $\theta_1$ is concerned, i.e., 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), i.e., $s$ is an instance of $t$, because we can obtain $s$ from $t$ by plugging in `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, these 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 $\theta$ such that ($\theta$ $s$) and ($\theta$ $t$) 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` $\mapsto$ `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 infinitely many monomorphic terms (or sentences).

## 2.15 A consequence of static scoping

Athena's procedural and deductive languages are both statically scoped. Roughly speaking, this means that the values of free identifier occurrences in procedures or methods are determined based on the textual structure of the code. This is in contrast to dynamic scoping, in which identifiers become bound to different values during execution as determined by the (dynamic) run-time stack of procedure calls rather than the (static) structure of the program text. Modern programming languages use static scoping in most cases 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`:

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

Procedure f defined.

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

Procedure g defined.

> (g 5)

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 as before? 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 procedure defined in line 1. This is a static 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 reenter 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

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 `||`.