```
> 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 `||`.

`true`, so we 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 $[t_1,\ldots,t_n]$ 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`) will take any sentence p and return 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 39), is carried out by the primitive ternary procedure `replace-var`. Specifically,

   (replace-var v t p)

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

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

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

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

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

2.17 Summary and notational conventions

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

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