```
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** construct 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 a lexical 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 lexical 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:
```

25 Recall that a lexical environment is a finite function mapping identifiers to values. Athena maintains a global lexical 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 lexical 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 here because it does not play a central role in the ideas we are discussing.

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

```
(not false)

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`, `right` $\mapsto$ `?s2`$\}$. Any occurrence of a term variable inside a pattern acts as a constant—it can only be matched by that particular variable. For example, the pattern `(S ?n)` will 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 $\{\texttt{s} \mapsto \texttt{ann}, \texttt{t} \mapsto \texttt{peter}\}$; 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

$$(\circ\ p_1 \cdots p_n) \tag{2.19}$$

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

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

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 (2.19) can also match a pattern of the form

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

when π is a list pattern. Here I will be bound to $\circ$ and π will be matched against the list $[p_1 \cdots p_n]$. 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 pc) (some-list args)) => [pc (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 lexical 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.