17.1. Tuple types

```
TupleType = "(" [ Type { "," Type } ] ")"
```

Dafny builds in record types that correspond to tuples and gives these a convenient special syntax, namely parentheses. For example, for what might have been declared as

```
datatype Pair<T,U> = Pair(0: T, 1: U)
```

Dafny provides the type `(T, U)` and the constructor `(t, u)`, as if the datatype's name were "" (i.e., an empty string) and its type arguments are given in round parentheses, and as if the constructor name were the empty string. Note that the destructor names are `0` and `1`, which are legal identifier names for members. For example, showing the use of a tuple destructor, here is a property that holds of 2-tuples (that is, *pairs*):

```
(5, true).1 == true
```

Dafny declares n-tuples where n is 0 or 2 or more. There are no 1-tuples, since parentheses around a single type or a single value have no semantic meaning. The 0-tuple type, `()`, is often known as the *unit type* and its single value, also written `()`, is known as *unit*.

18. Algebraic Datatypes

```
DatatypeDecl =
  ( "datatype" | "codatatype" )
  { Attribute }
  DatatypeName [ GenericParameters ]
  "=" [ ellipsis ]
      [ "|" ] DatatypeMemberDecl
      { "|" DatatypeMemberDecl }
      [ TypeMembers ]

DatatypeMemberDecl =
  { Attribute } DatatypeMemberName [ FormalsOptionalIds ]
```

Dafny offers two kinds of algebraic datatypes, those defined inductively (with `datatype`) and those defined co-inductively (with `codatatype`). The salient property of every datatype is that each value of the type uniquely identifies one of the datatype's constructors and each constructor is injective in its parameters.

18.1. Inductive datatypes

The values of inductive datatypes can be seen as finite trees where the leaves are values of basic types, numeric types, reference types, co-inductive datatypes, or function types. Indeed, values of inductive datatypes can be compared using Dafny's well-founded `<` ordering.

An inductive datatype is declared as follows:

```
datatype D<T> = _Ctors_
```

where *Ctors* is a nonempty `|`-separated list of *(datatype) constructors* for the datatype. Each constructor has the form:

```
C(_params_)
```

where *params* is a comma-delimited list of types, optionally preceded by a name for the parameter and a colon, and optionally preceded by the keyword `ghost`. If a constructor has no parameters, the parentheses after the constructor name may be omitted. If no constructor takes a parameter, the type is usually called an *enumeration*; for example:

```
datatype Friends = Agnes | Agatha | Jermaine | Jack
```

For every constructor `C`, Dafny defines a *discriminator* `C?`, which is a member that returns `true` if and only if the datatype value has been constructed using `C`. For every named parameter `p` of a constructor `C`, Dafny defines a *destructor* `p`, which is a member that returns the `p` parameter from the `C` call used to construct the datatype value; its use requires that `C?` holds. For example, for the standard `List` type

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

the following holds:

```
Cons(5, Nil).Cons? && Cons(5, Nil).head == 5
```

Note that the expression

```
Cons(5, Nil).tail.head
```

is not well-formed, since `Cons(5, Nil).tail` does not satisfy `Cons?`.

A constructor can have the same name as the enclosing datatype; this is especially useful for single-constructor datatypes, which are often called *record types*. For example, a record type for black-and-white pixels might be represented as follows:

```
datatype Pixel = Pixel(x: int, y: int, on: bool)
```

To call a constructor, it is usually necessary only to mention the name of the constructor, but if this is ambiguous, it is always possible to qualify the name of constructor by the name of the datatype. For example, `Cons(5, Nil)` above can be written

```
List.Cons(5, List.Nil)
```

As an alternative to calling a datatype constructor explicitly, a datatype value can be constructed as a change in one parameter from a given datatype value using the *datatype update* expression. For any `d` whose type is a datatype that includes a constructor `C` that has a parameter (destructor) named `f` of type `T`, and any expression `t` of type `T`,

```
d.(f := t)
```

constructs a value like `d` but whose `f` parameter is `t`. The operation requires that `d` satisfies `C?`. For example, the following equality holds:

```
Cons(4, Nil).(tail := Cons(3, Nil)) == Cons(4, Cons(3, Nil))
```

The datatype update expression also accepts multiple field names, provided these are distinct. For example, a node of some inductive datatype for trees may be updated as follows:

```
node.(left := L, right := R)
```

18.2. Co-inductive datatypes

TODO: This section and particularly the subsections need rewriting using the least and greatest terminology, and to make the text fit better into the overall reference manual.