implies that `0 <= n - i`, and gives Dafny a lower bound on the quantity. This also works when the bound `n` is not constant, such as in the binary search algorithm, where two quantities approach each other, and neither is fixed.

If the `decreases` clause of a loop specifies `*`, then no termination check will be performed. Use of this feature is sound only with respect to partial correctness.

19.13.3. Loop Framing

In some cases we also must specify what memory locations the loop body is allowed to modify. This is done using a `modifies` clause. See the discussion of framing in methods for a fuller discussion.

TO BE WRITTEN

19.14. Match Statement

```
MatchStmt =
  "match"
  Expression(allowLemma: true, allowLambda: true)
  ( "{" { CaseStmt } "}"
  | { CaseStmt }
  )

CaseStmt = "case" ExtendedPattern "=>" { Stmt }
```

[`ExtendedPattern` is defined in Section 20.32.]

The `match` statement is used to do case analysis on a value of an inductive or co-inductive datatype (which includes the built-in tuple types), a base type, or newtype. The expression after the `match` keyword is called the *selector*. The expression is evaluated and then matched against each clause in order until a matching clause is found.

The process of matching the selector expression against the `CaseBinding_`s is the same as for match expressions and is described in Section 20.32.

The code below shows an example of a match statement.

```
datatype Tree = Empty | Node(left: Tree, data: int, right: Tree)

// Return the sum of the data in a tree.
method Sum(x: Tree) returns (r: int)
{
  match x {
    case Empty => r := 0;
    case Node(t1, d, t2) =>
      var v1 := Sum(t1);
      var v2 := Sum(t2);
```

```
      r := v1 + d + v2;
  }
}
```

Note that the `Sum` method is recursive yet has no `decreases` annotation. In this case it is not needed because Dafny is able to deduce that `t1` and `t2` are *smaller* (structurally) than `x`. If `Tree` had been coinductive this would not have been possible since `x` might have been infinite.

19.15. Assert Statement

```
AssertStmt =
    "assert"
    { Attribute }
    ( [ LabelName ":" ]
      Expression(allowLemma: false, allowLambda: true)
      ( ";"
      | "by" BlockStmt
      )
    | ellipsis
      ";"
```

`Assert` statements are used to express logical proposition that are expected to be true. Dafny will attempt to prove that the assertion is true and give an error if the assertion cannot be proven. Once the assertion is proved, its truth may aid in proving subsequent deductions. Thus if Dafny is having a difficult time verifying a method, the user may help by inserting assertions that Dafny can prove, and whose truth may aid in the larger verification effort, much as lemmas might be used in mathematical proofs.

`Assert` statements are ignored by the compiler.

Using `...` as the argument of the statement is part of module refinement, as described in Section 21.

TO BE WRITTEN - assert by statements

19.16. Assume Statement

```
AssumeStmt =
    "assume"
    { Attribute }
    ( Expression(allowLemma: false, allowLambda: true)
    | ellipsis
    )
    ";"
```