The tnet Language

Table of Contents

Introduction

This document describes the **tnet language**. tnet is the task network language for the Systems Autonomy SR&TD Task (the Autonomy task).

tnet is a simple, statically-typed, domain-specific language for specifying task networks. Its goals are as follows:

1. To provide a convenient way to specify task networks as they are being developed in the Autonomy task.
2. To generate task networks for the MEXEC planning and execution software.
3. To generate C++ code that can be compiled into flight software (FSW), including the following:
    - C++ representations of the state data structures stored in a state database.
    - C++ code for translating between the state representation used by FSW and the state representation used by MEXEC.
    - Template instantiation functions that instantiate task templates into tasks on board.
    - C++ code for instantiating task templates based on real-time state information.

Overview

This section provides an overview of the tnet language. The following section provides a detailed description of the tnet language that is more rigorous and more complete than this section. It is probably best to read this section first. This section contains references to later sections of the manual that provide more information about the topics it discusses.

Values, Types, and Expressions

Values, types, and expressions are the basic elements of a tnet program.

Values

A **value** is one of the following:

- An **integer value** expressed in decimal notation (for example, `123`) or in hexadecimal notation (for example, `0xABCD`).
- A **floating-point value** expressed in standard notation, for example `1e-23`.
- A **string value**: a sequence of characters enclosed in double quotation marks, for example `"This is a string."`.
- A **Boolean value**: one of the two values `true` and `false`.
- A **structure value** representing a data structure with named members. For example, the value `{ x = 1, y = 2 }` represents a structure value with two members: the member `x` with value `1`, and the member `y` with value 2.
- An **array value** representing an array of values whose size is specified at compile time. For

example, the value `[ 1, 2, 3 ]` represents an array with the values 1, 2, and 3 as its elements. tnet arrays are zero-indexed.

- A **range value** representing all the values within a specified range. For example, the value `1..10` represents the integer values between 1 and 10, inclusive.
- A **set value** representing a set of values, specified as a collection of point values and ranges. For example, the value `set { 1, 2, 5..10 }` represents the values 1, 2, and 5 through 10.

Structure, array, range, and set values may contain structure and array values. For example, these are valid values:

```
{ x = { a = 1, b = 2 }, y = 0 }
[ { a = 1, b = 1 }, { a = 2, b = 2 }, { a = 3, b = 3 } ]
{ x = 1, y = 2 }..{ x = 2, y = 3 }
[ 0, 1 ]..[ 2, 3 ]
```

Ranges of structures and arrays are expanded elementwise. For example, `[ 0, 5 ]..[ 1, 6 ]` represents all values `[ x, y ]` such that `x` lies in the range `0..1` and `y` lies in the range `5..6`, i.e., the values `[ 0, 5 ]`, `[ 1, 5 ]`, `[ 0, 6 ]`, and `[ 1, 6 ]`.

Ranges and sets are not permitted as elements of structures or arrays. Strings are not permitted as elements of ranges or sets.

Structure, array, and set values have a flexible syntax: you can omit the commas and use line breaks instead. For example, these are legal values:

```
{
  x = 0
  y = 1
}
[
  0
  1
  2
]
set {
  0
  1
  10..11
}
```

Values must be **well-typed**; this is discussed further below. One rule of typing is that arrays, ranges, and sets must have elements with identical structure. As an example, the following array value is not well-typed, because the first element is an array value, and the second element is a structure value:

```
[ [ 0, 0 ], { x = 0 } ]
```

Types

Every value in tnet belongs to a **type**. A type is a class of values. tnet is a **statically-typed language**. That means that there are certain rules for computing the types of values, and those rules are checked at compile time, before any code is generated. If any rule violation is detected, then translation halts with an error message.

Each tnet value belongs to one of the following types:

- **Primitive integer types.** These are types of integer values. They are the familiar signed and unsigned integer types like `I32` and `U32`. `I` means "(signed) integer," and `U` means "unsigned integer." The number is the width of the representation in bits.
- **Primitive floating-point types.** These are the types of floating-point values. They are `F32` and `F64`, representing IEEE floating-point values of bit width 32 and 64.
- **The Boolean type.** This is the type of Boolean values. It is denoted with the keyword `boolean`.
- **The string type.** This is the type of string values. It is denoted with the keyword `string`.
- **Structure types.** A structure type is the type of a structure value. It looks like a structure value, but each member is followed by a colon and a type, instead of an equals sign and a value. For example, the type of the structure value `{ x = 1, y = 256 }` is `{ x : U8, y : U16 }`. Each member type of the structure type must be a structure member type.
- **Array types.** An array type is the type of an array value. It consists of a size enclosed in square brackets, followed by the element type. For example, the type of `[1 ,2, 3]` is `[3] U8`. You can read this as "array of three `U8`." The element type of an array type must be an array element type.
- **Enum types and named types.** An enum type is a user-defined type that denotes an enumeration, i.e., a set of discrete values. A named type is a name that refers to a type defined elsewhere. We will discuss these later, when we discuss type definitions and enum definitions.
- **Range types.** A range type is the type of a range value. It consists of the keyword `range` followed by the element type. For example, the type of `1..10` is `range U8`. The element type must be a range element type.
- **Set types.** A set type is the type of a set value. It consists of the keyword `set` followed by the element type. For example, the type of `set { 1, 2, 5..10 }` is `set U8`. The element type must be a set element type.

Structure, array, and set types have the same flexible syntax as structure, array, and set values. For example, this is a valid structure type:

```
{
  x : U32
  y : F32
}
```

tnet uses a simple rule for assigning types to nonnegative integer values: the type of an integer value is the narrowest type that will hold the value. For example, the type of the value 1 is `U8`, and the type of the value 256 is `U16`.

For negative integer values, tnet applies the following rule: compute the unsigned integer type of the absolute value, and use the next widest signed type. For example, the type of the value -1 is `I16`.

tnet assigns the type `F64` to floating-point values. To represent an `F32` literal value, you write an `F64` value and use an explicitly-typed expression to convert it to `F32` (see below).

Expressions

Expressions are units of syntax that are evaluated to values, either during translation or during program execution. Every expression has a type, and the type of the expression matches the type of the value to which it evaluates.

tnet provides the following kinds of expressions:

- **Value expressions.** Each of the values discussed above is an expression. For example, the literal values `1` and `1.23` are expressions. So are the structure value `{ x = 1, y = 2 }`, the array value `[ 1, 2, 3 ]`, the range value `1..10`, and the set value `set { 1, 2, 3 }`.
- **Negation expressions.** You can negate an expression of integer or floating-point type. For example, `-3` is a valid expression.
- **Binary arithmetic expressions.** You can combine numeric expressions using the standard arithmetic operators, and they have the standard precedence and meaning. For example, `1 + 2 * 3` is a valid expression, and it evaluates to `7`.

  You can use arithmetic operators to combine structures and arrays if both sides of the operators have the same shape, and the operator makes sense for the corresponding elements. For example, `{ x = 1, y = 1 } + { x = 2, y = 2 }` evaluates to `{ x = 3, y = 3 }`, `{ x = 1, y = 1 } * { x = 2, y = 2 }` evaluates to `{ x = 2, y = 2 }`, and `{ x = 2, y = 2 } / { x = 2, y = 2 }` evaluates to `{ x = 1, y = 1 }`.

- **Logical expressions.** You can use `and` to combine two Boolean expressions. The result is the logical and of the two subexpressions. For example, `true and false` evaluates to `false`.
- **Membership expressions.** You can use `in` to test set membership. For example `3 in { 1, 2, 3 }` returns `true`. The second expression must be a set or range that has the same element type as the first expression.
- **Equality expressions.** You can use `=` to test two expressions for equality. For example, `1 = 1` evaluates to `true`, while `1 = 2` evaluates to `false`.
- **Approximation expressions.** You can use `+-` (read "plus or minus") to represent an approximate value, expressed as a range. For example, `1 +- 0.1` evaluates to the range `0.9..1.1`.
- **Unevaluated subexpressions.** Wherever you can write a value as part of an expression, you can write a general expression (not necessarily a value) in the same place, so long as the types work out properly. The whole expression evaluates to a value in the obvious way. For example, `{ x = 1 + 2, y = 3 }` is a valid expression, and it evaluates to `{ x = 3, y = 3 }`.
- **Name expressions.** There are expressions for referring to qualified and unqualified names of definitions, such as `a` and `b`; we will discuss these later when we explain these definitions.
- **Structure member access expressions.** You can access a member of a structure by writing `.` and the member name after an expression of structure type. For example, `{ x = 1, y = 2 }.x`

evaluates to 1. You can apply the member access operator to a range or set expression; the result is the range or set constructed from the corresponding members. For example, `(a..b).x` evaluates to `a.x..b.x`, and `set { a, b }.x` evaluates to `set { a.x, b.x }`.

- **Array index expressions.** You can access an element of an array by writing an index expression enclosed in square brackets after an expression of array type. For example, `[ 1, 2, 3 ] [0]` evaluates to `1`. The index expression can be any expression that evaluates to a number. You can apply the array index operator to a range or set expression; the result is the range or set constructed from the corresponding array elements. For example, `(a..b)[0]` evaluates to `a[0]..b[0]`, and `set { a, b }[0]` evaluates to `set { a[0], b[0] }`.
- **Parenthesis expressions.** You can use parentheses to enforce evaluation order. For example, `(1 + 2) * 3` is a valid expression, and it evaluates to `9`.
- **Explicitly-typed expressions.** You can write an expression followed by a colon and a type. This is called an **explicitly-typed expression**, and it says to evaluate the expression and convert the result to the type. For example, `1 : F32` evaluates to the floating-point value `1.0` represented as an `F32` (4-byte floating-point value).

When typing and evaluating binary expressions with integer subexpressions, tnet widens the types of the subexpressions to ensure that there are enough bits to hold the result, until it gets to the widest type, which is either `I64` or `U64`. For example, `255 + 1` evaluates to `256:U16` and not `0:U8`.

Type Conversions

An explicitly-typed expression is an example of a **type conversion**, i.e., a reinterpretation of a value of one type as a value of another type. Some type conversions don't make sense: for example, `{ x = 1 } : F32` is illegal, because we can't make a structure value into an `F32` value. In general, the allowed conversions are the ones you would expect: we can convert from an integer or floating-point type to any other integer or floating-point type, and we can convert structures and arrays element by element, when the element types are convertible. We can also perform the following conversions:

- We can convert a value to a range, a value to a set, and a range to a set in the obvious way: for example, `1 : range U8` evaluates to `1..1`, `1 : set U8` evaluates to `set { 1 }`, and `(1..10) : set U8` evaluates to `set { 1..10 }`.
- We can convert an integer or floating-point value to a structure or array by assigning the value to each scalar element of the structure or array. For example, `0 : [2] { x : U32, y : U32 }` evaluates to `[ { x = 0, y = 0 }, { x = 0, y = 0 } ]`.

tnet automatically applies type conversions where it is necessary to satisfy the typing rules, and where it makes sense. For example, because `1 : [2] U8` is legal and evaluates to `[1, 1]`, the expression `[1, [2, 2]]` is legal and evaluates to `[[1, 1], [2, 2]]`. Similarly:

- `-1 * { x = 2, y = 2 }` evaluates to `{ x = -1, y = -1 } * { x = 2, y = 2 }`, which evaluates to `{ x = -2, y = -2 }`.
- `{ x = 1, y = 1 } +- 0.1` evaluates to `{ x = 1, y = 1 } +- { x = 0.1, y = 0.1 }`, which evaluates to `{ x = 0.9, y = 0.9 } .. { x = 1.1, y = 1.1 }`.

Further Details

For further details, see the following sections of this manual:

- Expressions
- Type Checking
- Evaluation

Defining Constants

The Basics

It is usually not a good idea to scatter literal constant values throughout a program. It is much better to give each constant value a name, and use the names. This makes the program more readable and maintainable.

In tnet, you use **constant definitions** to associate names with constant values. A constant definition consists of the keyword `constant`, a name, an equals sign, and an expression that evaluates to a compile-time constant value.

For example, the constant definition

```
constant ultimateAnswer = 42
```

declares the name `ultimateAnswer` and associates it with the value 42. Elsewhere in the program, you can write `ultimateAnswer` in place of the literal value 42.

Defining Several Constants

Most likely you will want to define more than one constant. tnet provides a flexible syntax for that. You can separate multiple constant definitions with line breaks, like this:

```
constant a = 1
constant b = 2
```

Alternatively, you can put multiple constant definitions on the same line. In this case, though, you have to use a semicolon to separate the definitions, like this:

```
constant a = 1; constant b = 2
```

Notice that newline characters separate constant definitions. Therefore, if you want to include a newline character within a constant definition, then you have to use the character `\`, like this:

```
constant a = \
0
```

If you omit the \ character in this case, you will get an error:

```
constant a = # Error
0
```

Complex Expressions

The expression on the right-hand side can be any valid expression. For example, here is a constant definition that uses a structure value to represent a vector with x, y, and z elements:

```
constant initialPosition = { x = 0 : F32, y = 0 : F32, z = 1 : F32 }
```

Here is an example that defines two constants and uses a binary arithmetic expression to define the second one in terms of the first one:

```
constant a = 1
constant b = a + 1
```

The order of definitions and uses of constants does not matter, as long as there are no cycles. For example, this is also perfectly legal:

```
constant b = a + 1
constant a = 1
```

Explicit and Implicit Types

You can write a type on the left-hand side, to specify the type of constant that you want. For example, you can write `initialPosition` this way:

```
constant initialPosition : { x : F32, y : F32, z : F32 } = { x = 0, y = 0, z = 1 }
```

If you don't write a type on the left, then the implied type of the constant is the type of the expression on the right. For example, the type of `initialPosition` here is `{ x : U8, y : U8, z : U8 }`:

```
constant initialPosition = { x = 0, y = 1, z = 1 }
```

Further Details

For further details, see the following sections of this manual:

- Constant Definitions
- Translation Units and Programs

Defining Types

The Basics

It is useful to be able to associate names with types. That way, you can use the names instead of the literal types. This makes the meaning of the program clearer. It also saves writing out complex types every time they are used.

In tnet, you use a **type definition** to associate a name with a type. A type definition consists of the keyword `type` followed by an identifier, an equals sign, and a type. For example:

```
type VectorF32 = { x : F32, y : F32, z : F32 }
```

The type created by the definition is called a **named type**.

Once you have defined a named type, you can use it instead of the type given on the right-hand side of the definition. For example:

```
constant initialPosition : VectorF32 = { x = 0, y = 0, z = 1 }
```

Writing Several Definitions

You can write several type definitions using the same syntax as for constant definitions. Definitions on the same line must be separated with semicolons, but definitions on separate lines don't require any separators. For example:

```
type t1 = U32; type t2 = I32
type t3 = F64
```

You can also interleave type definitions with constant definitions this way. For example:

```
type VectorF32 = { x : F32, y : F32, z : F32 }
constant initialPosition : VectorF32 = { x = 0, y = 0, z = 1 }
constant finalPosition : VectorF32 = { x = 1, y = 0, z = 0 }
constant a = 1; constant b = 2
```

Type Conversion

A type definition creates a new type. With one exception, described in the next section, that type is equivalent to the type on the right-hand side of the definition, called the **representation type**. In particular, you can freely convert back and forth between the defined type and the representation type. For example, this is allowed:

```
type t = F32
constant x : t = 0
```

It has the same meaning as this:

```
type t = F32
constant x : F32 = 0
```

Structure and Array Types

When tnet translators generate C++ code, structure and array types become classes. In order to provide meaningful names to these classes, we place the following restrictions on type definitions:

1. No structure type appearing on the right-hand side of a type definition may have a structure or array type for any of its member types.
2. No array type appearing on the right-hand side of a type definition may have a structure or array type as its element type.

```
type t = { x : { y : U32 } } # Error
```

This type definition violates the rule: it defines a named structure type with a member `x` whose structure type has no name.

If you want to use a structure or array type inside another structure or array type on the right-hand side of a type definition, then you need to provide names for both types. For example:

```
type t1 = { y : U32 } # OK
type t2 = { x : t1 } # OK
```

Further Details

For further details, see the following sections of this manual:

- Type Definitions
- Translation Units and Programs
- Type Checking

Defining Enumerations

The Basics

It is useful to be able to define a new type that represents a fixed set of values. For example, the values can be a set of distinct numbers, each one of which represents a different body vector of a spacecraft.

In tnet, you do this with an **enum definition** (short for enumeration definition). An enum definition defines a type that represents a fixed set of constant integer values. The values are called the **enumerated constants** of the enum.

To write an enum definition, you write the keyword `enum` followed by a name and a list of enumerated constants enclosed in braces. Each enumerated constant has the form *name* `=` *expression*. For example:

```
enum BodyVector {
  ANTENNA = 0
  CAMERA_BORESIGHT = 1
  SOLAR_PANEL = 2
}
```

Elsewhere in the program, you can use the name of the enum as a type. For example:

```
type t = BodyVector
```

This code uses a type definition to associate the name `t` with the type `BodyVector`.

You can also refer to the enumerated constants. You have to qualify them with the enum name using a dot. For example:

```
constant initialBodyVector = BodyVector.SOLAR_PANEL
```

The Representation Type

Every enum definition has an associated integer type called the **representation type** of the enum. When translating a tnet source program into an MEXEC or C++ representation, the enumerated constant values become integer values of the representation type.

The tnet semantic analyzer infers a representation type from the types of the enumerated constants. To do so, it uses the following rules:

- If any of the enumerated constants is a signed integer, then the representation type is a signed integer. Otherwise the representation type is an unsigned integer.
- The representation type has the smallest width that can hold all the values of the enumerated constants, if there is one. Otherwise the width is 64.

For example, the inferred representation type of the enum `BodyVector` defined above is `U8`.

Optionally, you can provide an explicit representation type, as follows:

```
enum BodyVector : I32 {
  ANTENNA = 0
  CAMERA_BORESIGHT = 1
  SOLAR_PANEL = 2
}
```

In this case, the representation type is `I32`.

Both the inferred representation type and the explicit representation type (if there is one) must be integer types.

Writing Several Definitions

You can write several enumerated constants and multiple enum definitions using the same syntax as for constant definitions and type definitions. For example:

```
enum E1 { X = 0, Y = 1 }; enum E2 { A = 0, B = 1 }
enum E3 { C = 0, D = 1 }
```

Notice that the required separator for enumerated constants appearing on the same line is a comma.

You can also interleave enum definitions with type definitions and constant definitions this way. For example:

```
enum E { X = 0, Y = 1 }
type t = E
constant c : t = E.X
```

Type Conversion

Unlike a type definition, an enum definition defines a new type. A value of the type may be converted to a value of integer or floating point type. For example, this is legal:

```
enum E { X = 0 }
constant c : U8 = E.X
```

In this case, the constant `c` represents the value `0` with type `U8`. However, values of other types, including numeric types, may *not* be converted to values of enum type. For example, this is illegal:

```
enum E { X = 0 }
constant c : E = 1 # Error
```

tnet does not let you convert the value `1` of type `U8` to type `E`.

Further Details

For further details, see the following sections of this manual:

- Enum Definitions
- Enumerated Constant Definitions
- Translation Units and Programs
- Type Checking

Defining Modules

The Basics

For modular programming, it is important to use hierarchical naming, that is, to place groups of names inside scopes that are qualified with other names. For example, suppose you are defining a system with two subsystems `Heater` and `Thruster`, and each subsystem has a constant called a control parameter. You can't name both constants `controlParameter` in the same scope, because the names will clash. Without hierarchical naming, you might name the two parameters `heaterControlParameter` and `thrusterControlParameter`. This works, but it's better to define two scopes, one for `Heater` and one for `Thruster`, and to declare a constant `controlParameter` in each one. Then you can refer to the parameters as `Heater.controlParameter` and `Thruster.controlParameter`.

In tnet, you do this with a **module definition**. To write a module definition, you write the keyword `module` followed by a module name and a set of definitions enclosed in braces. For example:

```
module Heater {
  constant controlParameter = 0.001
}
```

Writing Several Definitions

At the top level, you can write several module definitions using the same syntax as for constant definitions, type definitions, and enum definitions. You can also interleave module definitions and other definitions in this way. For example:

```
type ParameterType = F32
module Heater {
  constant controlParameter = 0.001 : ParameterType
}
module Thruster {
  constant controlParameter = 5.678 : ParameterType
}
```

Inside a module definition, you write other definitions (including other module definitions), using the same syntax as for writing definitions at the top level. For example:

```
type ParameterType = F32
module Heater {
  constant controlParameter = 0.001 : ParameterType
  constant scaleFactor = 10 : F32
}
module Thruster {
  constant controlParameter = 5.678 : ParameterType
  constant scaleFactor = 100 : F32
}
```

Referring to Definitions Enclosed in Modules

Inside a module definition, you can use the unqualified names of definitions appearing there. Outside the definition, you have to use the qualified name. For example:

```
type ParameterType = F32
module Heater {
  constant controlParameter = 0.001 : ParameterType
  constant scaleFactor = 10 : F32
  constant scaledControlParameter = controlParameter * scaleFactor
}
module Thruster {
  constant controlParameter = 5.678 : ParameterType
  constant scaleFactor = 100 : F32
  constant scaledControlParameter = controlParameter * scaleFactor
}
constant averageScaleFactor = (Heater.scaleFactor + Thruster.scaleFactor) / 2
```

Reopening Module Scopes

You can close a module scope and reopen it later. For example, this program is legal:

```
module M {
  constant a = 0
}
module M {
  constant b = 1
}
```

It is equivalent to the following program:

```
module M {
  constant a = 0
  constant b = 1
}
```

Reopening scopes is useful, for example, if you want to split the definition of a module across several different files.

Further Details

For further details, see the following sections of this manual:

- Module Definitions
- Translation Units and Programs

Writing Comments and Annotations

The Basics

You can write a **comment** that appears in the source program but is ignored by the translator. Comments start with the character `#` and go to the end of the line. For example:

```
# This is a comment.
```

You can also write an **annotation** that has no meaning in tnet but is carried through the translation (for example, it may become a comment in generated C++ code). The following syntax elements may have annotations:

- Definitions.
- Parameters of commands and task templates.
- The elements of a structure type.

There are two kinds of annotations: **pre-annotations** and **post-annotations**.

- A pre-annotation starts with the character `@` and is attached to the element of syntax that follows it.

- A post-annotation starts with the characters `@<` and is attached to the element of syntax that precedes it.

In each case, the annotation goes to the end of the line. For example:

```
@ This is a pre-annotation
enum E {
  X = 0 @< This is a post-annotation
}
```

Multiline annotations are allowed. For example:

```
@ This is a pre-annotation
@ It has two lines.
enum E {
  X = 0 @< This is a post-annotation
        @< It has two lines.
}
```

Further Details

For further details, see the following sections of this manual:

- Comments and Annotations

Defining State Data

The Basics

tnet supports autonomous flight software systems in which data representing the system state is stored in a data structure called the **state database**. The state database partitions the state data into several kinds, and each kind of data has an associated structure type. For example, there might be guidance, navigation, and control (GNC) data, imager data, and power data, with a separate data structure for each kind of data. Within each structure, the members hold individual components of the state data for that kind of state. For example, the GNC data type might have an element that stores the current estimate of spacecraft position.

In tnet, you define state data with a **state data definition**. A state data definition consists of the keyword `state`, an identifier, an equals sign, and an expression that evaluates to a compile-time constant structure value. The identifier names the state data. The value on the right-hand side represents the type and the initial value of the state data.

For example, in the following code, the state definition `gncState` represents GNC state containing an attitude quaternion and a position vector. The initial attitude quaternion is `{ q1 = 0, q2 = 0, q3 = 0, q4 = 1 }`, and the initial position vector is `{ x = 0, y = 0, z = 1 }`.

```
type QuaternionF32 = { q1 : F32, q2 : F32, q3 : F32, q4 : F32 }
type VectorF32 = { x : F32, y : F32, z : F32 }
type GNCState = {
  attitudeQuaternion : QuaternionF32
  positionVector : VectorF32
}
state gncState = {
  attitudeQuaternion = { q1 = 0, q2 = 0, q3 = 0, q4 = 1 }
  positionVector = { x = 0, y = 0, z = 1 }
} : GNCState
```

The expression on the right-hand side must be of **state data type**. A state data type is either a named type whose representation type is a structure type or a named type whose representation type is a state data type.

Explicit and Implicit Types

Optionally, you can write a type on the left-hand side. For example, you can write `gncState` this way:

```
state gncState : GNCState = {
  attitudeQuaternion = { q1 = 0, q2 = 0, q3 = 0, q4 = 1 }
  positionVector = { x = 0, y = 0, z = 1 }
}
```

Any type appearing on the left has to be a named type. Notice that if there is a named type on the left, then the type on the right can be a structure type; it doesn't have to be a named type. If you don't write a type on the left, then the implied type of the state data is the type of the expression on the right, and the type of the expression must be a named type.

Writing Several Definitions

You can write several state data definitions using the same syntax as for constant definitions and type definitions. You can also interleave state data definitions with other definitions this way. For example:

```
type S1 = { a : U8, b : U8 }
type S2 = { c : F32, d : F32 }
type S3 = { e : U32, f : U32 }
state s1 : S1 = { a = 1, b = 2 }; state s2 : S2 = { c = 3, d = 4 }
state s3 : S3 = { e = 5, f = 6 }
```

Further Details

For further details, see the following sections of this manual:

- State Data Definitions

- Translation Units and Programs

Defining Planning Timelines

The Basics

A **timeline** is a state variable maintained by the planner. The planner uses timelines to predict the future values of state variables as it constructs schedules. For example, a timeline might be a state variable `x` of type `U32` with initial value 0. If the planner schedules a task `T` whose effect on the state is to set `x` to 1, then the planning timeline for `x` records the fact that the value of `x` is 0 until `T` has finished running, at which point it changes to 1.

To specify a timeline in tnet, you write the word `timeline` followed by a member of a state data structure. For example:

```
type S = { x : U16, y : U32 }
state s : S = 0
timeline s.x
```

In this example, the state data structure `s` has two members, `x` and `y`. `s.x` is a timeline, but `s.y` is not.

Restricting the Values of a Timeline

In the example above, the timeline `s.x` can attain all legal values of the `U16` type. You can also write that explicitly using a range expression, as follows:

```
type S = { x : U16, y : U32 }
state s : S = 0
timeline s.x in 0..65535
```

If you wish, you can restrict the range to a subset of the `U16` values, as follows:

```
type S = { x : U16, y : U32 }
state s : S = 0
timeline s.x in 0..40000
```

Finally, you can use a set expression instead of a range expression after the `in` keyword:

```
type S = { x : U16, y : U32 }
state s : S = 0
timeline s.x in set { 0, 10, 10000..40000 }
```

This says that the values of `s.x` are restricted to be 0 or 10 or to lie in the range `10000..40000`.

Structure and Array Timelines

A timeline can have a structure or array type. For example, this is legal:

```
type VectorF32 = { x : F32, y : F32 , z : F32 }
type State = { v1 : VectorF32, v2 : VectorF32 }
state s : State = 0
constant min : VectorF32 = -1
constant max : VectorF32 = 1
timeline s.v1 in min..max
```

Restrictions on Timeline Types

The type of a timeline may not be `string`, a range type, or a set type. If a timeline has a named type that resolves to a structure or array type, then none of the members of the structure or array (or members of the members, and so on) may have type `string` or a range or set type.

For example, this code is legal:

```
type State = { x : F32, s : string }
state s : State = { x = 0, s = "" }
timeline s.x
```

However, this code is not legal, because it attempts to make a string member into a timeline:

```
type State = { x : F32, s : string }
state s : State = { x = 0, s = "" }
timeline s.y # Illegal
```

Similarly, this code is illegal, because the type `t` has a string member:

```
type t = { s : string }
type State = { x : F32, y : t }
state s : State = { x = 0 , y = { s = "" } }
timeline s.y # Illegal
```

Members of Members

You can make a member of a member of a state data structure into a timeline, to any depth. For example, this is legal:

```
type S = { a : { x : U32, y : [2] U8 }, b : U16 }
state s : S = 0
timeline s.a.y[0]
```

Enumerated Timelines

If a timeline has an enumeration type, then by default it can attain all the enumerated constants of the enumeration. For example, in this code

```
enum E { X = 0, Y = 1, Z = 2 }
type S = { a : E }
state s : S = { a = E.X }
timeline s.a
```

the timeline `s.a` may attain the values `E.X`, `E.Y`, and `E.Z`. If you want to restrict the values to a subset of the enumerated constants, then you must use a set. For example:

```
enum E { X = 0, Y = 1, Y = 2 }
type S = { a : E }
state s : S = { a = E.X }
timeline s.a in set { E.X, E.Y }
```

Ranges of enumerated values are not allowed. For example:

```
enum E { X = 1, Y = 10 }
type S = { a : E }
state s : S = { a = E.X }
timeline s.a in E.X..E.Y # Error; range of enum not allowed
```

Writing Several Timelines

You can write several timelines using the same syntax as for other elements such as constant definitions. For example:

```
timeline s.a
timeline s.b; timeline s.c
```

As shown in the examples above, you can also interleave timeline definitions with other definitions this way.

Further Details

For further details, see the following sections of this manual:

- Timeline Definitions
- Translation Units and Programs

Defining Commands

The Basics

tnet supports task networks in which tasks issue commands. Therefore, we need a way to specify commands. We do that with a **command definition**.

A command definition specifies the name of a command, an optional list of typed parameters, and a numeric opcode. For example, the following code defines a command called `NoOp` with arguments `arg1` of type `F32` and `arg2` of type `U32` and opcode `0x1234`:

```
command NoOp(arg1 : F32, arg2 : U32) opcode 0x1234
```

Each type appearing in a command parameter must be a primitive type, an enum type, a string type, or a named type that refers to one of those types.

Defining Several Commands

You can write several command definitions using the same syntax as for constant definitions and type definitions. You can also interleave command definitions with other definitions this way.

Further Details

For further details, see the following sections of this manual:

- Command Definitions

Defining Task Templates

The Basics

In tnet, a **task** is a fundamental unit of execution. A task consists of a spacecraft command together with some metadata specifying the conditions under which the command should be issued.

Tasks may be constructed from **task templates**. A task template specifies part of a task. The rest of the task is specified when the template is instantiated to a task and included in a task network.

In tnet, there are two kinds of task templates:

1. A with task template directly specifies the command and the metadata that define the template.
2. A from task template names a template defined elsewhere and specifies additional information.

A task template can have parameters, which are unbound values used in the body of the template. The values become bound when the template is instantiated.

With Task Templates

A **with task template** directly specifies the elements of a task template. It consists of the keywords `task template`, the name of the template, the keyword `with`, and a **task body** enclosed in braces. A task body is a list of elements, each of which may be terminated by a semicolon or a newline.

As an example, here is a simple with task template that specifies a no-op command:

```
task template NoOp with {
  command Diagnostic.NO_OP()
}
```

If desired, we can omit an empty argument list `()` to a command. For example, this is also allowed:

```
task template NoOp with {
  command Diagnostic.NO_OP
}
```

Here is a with task template that turns on an imager:

```
task template ImagerOn with {
  pre Power.imagerState = OnOff.OFF
  impact Power.imagerState = OnOff.ON
  command Power.IMAGER(OnOff.ON)
}
```

The body of the template contains the following elements:

- A **precondition** specifying that in order for this task to be executed, the value of `Power.imagerState` must be `OnOff.OFF`. Here, `Power.imagerState` is a timeline.
- An **impact** specifying that after this command is executed, the state of `Power.imagerState` is `OnOff.ON`.
- A **command** specifying what command to issue when executing the task. Here the command is `Power.IMAGER`, and it has one argument `OnOff.ON`.

For a complete list of elements that may appear in a task template body, see the section on task bodies.

From Task Templates

A **from task template** constructs a new task template T' from another task template T defined elsewhere. It consists of the keywords `task template`, the name of the template, the keyword `from`, and the name of a task template *T*, the keyword `with`, and a **task body** *B* enclosed in braces. This construct creates a new template by taking the definition of *T* and adding the elements appearing in *B*. If any element appears in both *T* and *B*, and only one such element is allowed in a task body, then the element appearing in *B* supersedes the element appearing in *T*.

As an example, here is a from task template that takes the `NoOp` template defined in the previous section and adds information on when to schedule the task and how long the task should take:

```
task template NoOpWithTimes from NoOp with {
  start 100
  duration 1
}
```

This template refers to the `NoOp` template and inherits the specification of that template. In addition, it specifies that the task should be scheduled to start at time 100 and is expected to run for one second.

When writing a from template, you can omit the `with` clause and the task body. As discussed in the next section, this feature is useful in conjunction with task template parameters.

Task Template Parameters

A task template can have parameters that become bound to arguments when the template is used. As an example, here is a template called `ImagerPower` that does not specify whether we are turning the imager on or off.

```
task template ImagerPower(
    preState : OnOff
    postState : OnOff
) with {
  pre Power.imagerState = preState
  impact Power.imagerState = postState
  command Power.IMAGER(postState)
}
```

The template has two parameters:

1. `preState` specifies the expected imager power state when the task is run.
2. `postState` specifies the new state as a result of running the task.

When using a template with parameters, you must provide specify arguments that bind values to the parameters, like this:

```
task template ImagerOn from ImagerPower(
    OnOff.OFF
    OnOff.ON
) with {
  start 100
  duration 1
}
```

Here we have bound the value `OnOff.OFF` to the `preState` parameter and the value `OnOff.ON` to the

`postState` parameter. We have used that template, with the bound value, to construct a new template `ImagerOn`.

As mentioned in the previous section, you can omit the `with` clause. For example, you can write this:

```
task template NoOp(
    startArg : U32
    durationArg : U32
) with {
  start startArg
  duration durationArg
  command Test.NO_OP
}

task template NoOpWithTimes from A(100, 1)
```

The template `NoOp` specifies a command with a `NO_OP` command and unbound start time and duration. The template `NoOpWithTimes` binds the values 100 and 1 the the start time and the duration.

Defining Several Task Templates

You can write several task templates using the same syntax as for constant definitions . You can also interleave task template definitions with other definitions in this way.

Further Details

For further details, see the following sections of this manual:

- Task Template Definitions
- Parameter Lists
- Argument Lists
- Task Bodies

Defining Task Networks

The Basics

A **task network** is a collection of tasks. To write a task network, you write the keyword `tasknet` followed by the name of the task network and a **task network body** enclosed in braces. The task network body is a list of elements, each of which is terminated by a newline or a semicolon.

The following elements may appear in a task network body:

1. A timeline element. Including a timeline element *T* in the task network tells the planner to create *T* when the task network is loaded.
2. A with task element. A with task element directly defines a task to be included in the task network.

3. A from task element. A from task element instantiates a task *T* from a task template and includes *T* in the task network.
4. A task template element. A task template element includes an uninstantiated task template in the task network, so that the planner can instantiate it.

Timeline Elements

A **timeline element** creates a timeline in the planner. To write a timeline element, you write the keyword `timeline` followed by a qualified identifier that refers to a timeline definition. Here is an example:

```
type S = { x : U16, y : U32 }
state s : S = 0
timeline s.x

tasknet T {
  timeline s.x
}
```

The first three lines define a structure type `S`, a state variable `s` of type `S`, and a timeline `s.x`.

The task network definition defines a task network `T`. It refers to the timeline `s.x`. When loaded into the planner, this task network will cause the timeline `s.x` to be created.

A timeline element must refer to a valid timeline. For example, this code is invalid, because `s.y` is not a timeline:

```
type S = { x : U16, y : U32 }
state s : S = 0
timeline s.x

tasknet T {
  timeline s.y # Error
}
```

With Task Elements

A **with task element** directly specifies the elements of a task. It consists of the keyword `task`, an identifier, the keyword `with`, and a task body enclosed in braces. The identifier names the task. The task body specifies the elements of the task. It has the same syntax as the body of a task template.

Here is an example:

```
tasknet T {

  task ImagerOn with {
    pre Power.imagerState = OnOff.OFF
    impact Power.imagerState = OnOff.ON
    command Power.IMAGER(OnOff.ON)
  }

  task TakeImage with {
    pre Power.imagerState = OnOff.ON
    command Imager.TAKE_IMAGE
  }

}
```

This is a task network with two tasks. The first task turns on an imager. Its precondition is that the imager is off, and its impact is that the imager is on. The second task takes an image. Its precondition is that the imager is on.

Loading Timelines in MEXEC

Note that the MEXEC planner is rather particular about the way that timeline elements appear in task networks.

First, when you load a task network *T* into the planner, any timelines used in the task must either (1) be included as timeline elements in *T* or (2) already exist in the planner as a result of loading a previous task network. For example, if you started with the planner in its initial state and loaded just the task network shown in the previous section, you would get a runtime error. The task network shown above assumes that the timeline `Power.imagerState` has already been loaded into the planner as part of a different task network. For example, you could load this task network first:

```
tasknet Timelines {

  timeline Power.imagerState

}
```

Second, once a timeline *t* has been loaded into the planner, loading *t* again causes a runtime error. For example, given the task network `Timelines` shown above, loading `Timelines` twice would work the first time and fail the second time.

For this reason, it is usually a good idea to include timeline elements in separate task networks from task elements. That way, you can load a single set of the timelines once and then load several task networks against the timelines.

From Task Elements

A **from task template** consists of the keyword `task`, the name of the task, the keyword `from`, the name of a task template, and an optional `with` clause that specifies additional elements. Here is an example:

```
module Templates {

  task template ImagerPower(powerState : OnOff) with {
    pre Power.imagerState = powerState
    impact Power.imagerState = powerState
    command Power.IMAGER(powerState)
  }

  task template TakeImage with {
    pre Power.imagerState = OnOff.ON
    command Imager.TAKE_IMAGE
  }

}

tasknet Imaging {

  task ImagerOn from Templates.ImagerPower(OnOff.ON) with {
    start 100
    duration 1
  }

  task TakeImage from Templates.TakeImage

}
```

This example defines two task templates, `Templates.ImagerPower` and `Templates.TakeImage`. The task network `Imaging` instantiates each of the templates into a from task. The first contains has a `with` clause that specifies the start time and duration of the task. The second task omits the `with` clause.

Task Template Elements

A **task template element** consists of the keywords `task template` followed by a qualified identifier that names a task template and an optional list of arguments. The arguments specify the default values to use when instantiating the template. When a task network containing a task template element *E* is loaded into the planner, *E* becomes a task template that the planner can instantiate as necessary when constructing plans.

Here is an example:

```
module Templates {

  task template ImagerPower(powerState : OnOff) with {
    pre Power.imagerState = powerState
    impact Power.imagerState = powerState
    command Power.IMAGER(powerState)
  }

  task template TakeImage with {
    pre Power.imagerState = OnOff.ON
    command Imager.TAKE_IMAGE
  }

}

tasknet Imaging {

  task template Templates.ImagerPower(OnOff.ON)

  task TakeImage from Templates.TakeImage

}
```

In this example, the task network `Imaging` contains one template and one task `TakeImage`. When this task network is loaded into the planner, the planner can instantiate the template `Templates.ImagerPower` into a task and schedule that task. Executing that task turns the imager on. Once the imager is on, the planner can schedule the task `TakeImage`, whose precondition requires that the imager is on.

Optionally, you can write a `with` clause at the end of a task template element. If present, the with clause operates similarly to a with clause in a template definition. It adds elements to the template. For example, here is a second version of the `Imaging` task network in which the task template element contains a with clause:

```
tasknet Imaging {

  task template Templates.ImagerPower(OnOff.ON) with {
    start 100
    duration 1
  }

  task TakeImage from Templates.TakeImage

}
```

The with clause specifies the start time and the duration associated with the template.

Defining Several Task Networks

You can write several task networks using the same syntax as for constant definitions. You can also interleave task network definitions with other definitions in this way.

Further Details

For further details, see the following sections of this manual:

- Task Template Definitions
- Parameter Lists
- Argument Lists
- Task Bodies
- Task Network Definitions

Writing and Translating Programs

The Basics

A **program** is a complete unit of tnet source that can be analyzed and translated. It can consist of several files.

tnet is designed to be flexible and to support multiple code generation strategies. Each strategy has its own translation tool, called a **translator**. Currently the following tools exist:

- `tnet-check`: Check the validity of a tnet program, without performing any translation.
- `tnet-constants`: Extract constant definitions for use by MEXEC.
- `tnet-enums`: Extract enum definitions for use in FSW code.
- `tnet-arrays`: Extract array type definitions for use in FSW code.
- `tnet-structs`: Extract structure type definitions for use in FSW code.
- `tnet-states`: Extract state definitions for use in FSW code.
- `tnet-timelines`: Extract timeline definitions for use by MEXEC.
- `tnet-state-translator`: Extract timeline definitions for use in MEXEC C++ state translators.

Using the Tools

See here for information on how to use these tools.

Further Details

For further details, see the following sections of this manual:

- Translation Units and Programs
- Translation

Detailed Description

This section provides a detailed description of the tnet language. We fully describe each syntactic element of the language. We also describe semantics as appropriate. Additional general semantic rules are described in subsequent sections.

Language Elements: We divide the language syntax into the following elements:

- Lexical Elements: The lexical elements of a tnet program.
- Identifiers: The names of the definitions and parameters that are defined and used in a tnet program.
- Types: The types of the expressions and values in a tnet program.
- Expressions: Syntactic units that have types and that evaluate to values.
- Definitions: Definitions appearing in a tnet program. Module definitions themselves contain definitions, including definitions of other modules. Enum definitions contain definitions of enumerated constants.
- Element Sequences: Sequences of elements, for example, all the definitions appearing in a module.
- Comments and Annotations: Program text that is not parsed and is there solely for human users to read.
- Translation Units and Programs: Groupings of definitions that can be checked for semantic correctness and translated.

Syntax Notation: We use the following notation to describe the syntax of language elements:

- `fixed-width font` denotes literal program text.
- *italics* denote grammar nonterminals.
- Italicized brackets *[ ]* enclose optional elements.

Lexical Elements

Before parsing a tnet program, the compiler converts the source text into a list of tokens. This process is called lexing.

Tokens

A token is one of the following:

- A reserved word.
- A symbol.
- An identifier.
- An integer literal.
- A floating-point literal.

- A Boolean literal.
- A string literal.
- A line of an annotation beginning with `@` or `@<`.

Comments

Comments are ignored during lexing.

Whitespace

The lexer treats whitespace as follows:

- Space characters are ignored, except to separate tokens.
- The newline character is a token. It is used to separate the elements of element sequences.
- No other whitespace characters are allowed. In particular, the tab character may not appear (outside of a comment, annotation, or string literal) in a tnet program.

Line Continuations

The character `\`, when appearing before a newline, causes the newline to be ignored. For example, this

```
constant a = 1 + \
  1
```

is equivalent to this:

```
constant a = 1 + 1
```

Note that the `\` character is required in this case. For example, the following is not syntactically correct:

```
constant a = 1 + # Error
  1
```

The newline indicates the end of an expression, but `1 +` is not a valid expression.

The lexer ignores any space characters appearing between `\` and the next newline.

Reserved Words

The following are reserved words in tnet. They may not be used as identifiers.

```
F32
F64
I16
I32
I64
I8
U16
U32
U64
U8
and
boolean
cleanup
command
constant
contingency
duration
end
enum
false
from
impact
in
maint
module
opcode
planning
post
pre
range
rate
set
start
state
string
task
tasknet
template
timeline
true
type
with
```

Symbols

The following sequences of characters are symbol tokens in tnet:

```
*
+
+-
+=
,
-
.
..
/
:
;
=
[
]
{
}
```

Identifiers

An **identifier** is the unqualified name of a structure member, a definition, or a parameter.

Syntax

An identifier consists of one or more characters. The first character must be a letter or an underscore. Characters after the first character may be letters, digits, and underscores.

Examples

```
identifier1
Identifier2
thisIsIdentifier3
ThisIsIdentifier4
this_is_identifier_5
_identifier6
```

Types

Primitive Types

Primitive types are the types of the primitive scalar values supported by most general-purpose computers. There are four kinds of primitive types: unsigned integer types, signed integer types, floating-point types, and the Boolean type.

- The unsigned integer types are `U8`, `U16`, `U32`, and `U64`. The `U` stands for “unsigned.” The number after the U is the width of the representation in bits, using the standard binary representation of an unsigned integer.

- The unsigned integer types are `I8`, `I16`, `I32`, and `I64`. The `I` stands for "[signed] integer." The number after the I is the width of the representation in bits, using the standard binary two's complement representation of a signed integer.
- The floating-point types are `F32` and `F64`. These are IEEE floating-point values of width 32 and 64 bits, respectively.
- The Boolean type is `boolean`. This type represents the two values `true` and `false`.

The signed and unsigned integer types, collectively, are called the **integer types**. The integer types together with the floating-point types are called the **numeric types**.

The String Type

The type `string` is the type of string values.

Enum Types

An **enum type** is a qualified identifier that refers to an enum definition via the rules for name scoping. For example:

```
module M {
  enum E { X = 0, Y = 1 }
}
constant a = M.E.X # a has type M.E
type t = M.E # t is another name for M.E
constant b : t = M.E.Y # b has type t
```

`M.E` is an enum type.

Structure Types

A **structure type** represents a data structure with members. Each member has a name and a type. The order of the members appearing in the type is not significant. Two or more members with the same name are not allowed.

The types of the structure members are called the **member types** of the structure type. Each member type of a structure type must be an structure member type.

Syntax

{ *structure-type-member-sequence* }

structure-type-member-sequence is an element sequence in which each element is a structure type member of the form *identifier* : *type*, and the terminating punctuation is a comma.

Semantics

identifier is the name of the member, and *type* is its type.

Example

`{ x : U8, y : F32 }` is a structure type with two members, `x` of type `U8` and `y` of type `F32`. We can also write this type as follows:

```
{
  x : U8
  y : F32
}
```

Structure Member Types

A **structure member type** is the type of a member of a structure type. It is a primitive type, the string type, an enum type, a structure type whose members all have structure member type, an array type whose element type is a structure member type, or a named type that refers to a structure member type.

Examples

- `U32` is a structure member type.
- `range U32` is not a structure member type.
- `{ x : U32 }` is a valid structure type.
- `{ x : range U32 }` is not a valid structure type, because `range 32` is not a structure member type.
- `{ x : { y : U32 } }` is a valid structure type.
- `{ x : { y : range U32 } }` is not a valid structure type because `{ y : range U32 }` is not a structure member type (and also not a valid type).

Array Types

An **array type** represents an indexed collection of values.

Syntax

`[` *expression* `]` *type*

Semantics

expression represents the size of the array. Its type must be a numeric type, and it must evaluate to a compile-time constant.

type is called the **element type** of the array type. It represents the type of each array element. It must be an array element type.

Example

```
constant a = [ 0, 1 ] # Type is [2] U8
constant b = [ { x = 0, y = 1 }, { x = 2, y = 3 } ] # Type is [2] { x : U8, y : U8 }
constant c = [ [ 0, 1 ], [ 2, 3 ], [ 4, 5 ] ] # Type is [3] [2] U8
```

In the last constant definition, note that the array type reads naturally from left to right as "array of 3 array of 2 U8."

Array Element Types

An **array element type** is the element type of an array type. A type T is an array element type if and only if T is a structure member type.

Examples

- `U32` is an array element type.
- `range U32` is not an array element type.
- `[10] U32` is a valid array type.
- `[10] range U32` is not a valid array type, because `range 32` is not an array element type.
- `[10] [20] U32` is a valid array type.
- `[10] [20] range U32` is not a valid array type because `[20] range U32` is not an array element type (and also not a valid type).

Range Types

A **range type** represents a range of values.

Syntax

`range` *type*

Semantics

type is called the **element type** of the range type. It represents the type of each element of the range. It must be an range element type.

Examples

- The type of the expression `0..1` is `range U8`.
- The type of the expression `[ 0, 1 ]..[ 1, 2 ]` is `range [2] U8`.
- The type of the expression `{ x = 0.0, y = 0.0 }..{ x = 1.0, y = 1.0 }` is `range { x : F64, y : F64 }`.
- The following code is invalid, because `E` is not a range element type:

```
enum E { A = 0, B = 1, C = 2, D = 3 }
constant r = E.A..E.C # Invalid: E is not a range element type
```

Range Element Types

A **range element type** is the element type of a range type. It is a numeric type, a structure type whose members all have range element type, an array type whose element type is a range element type, or a named type that refers to a range element type.

Examples

- `U32` is a range element type.
- `range U32` is a valid range type but not a range element type.
- `range range U32` is not a valid range type, because `range 32` is not a range element type.
- `range string` is not a valid range type, because `string` is not a range element type.
- `range { x : U32 }` is a valid range type.
- `range { x : string }` is not a valid range type because `{ x : string }` is not a range element type.
- `range { x : range U32 }` is not a valid range type because `{ x : range U32 }` is not a range element type (and also not a valid type).
- This code illustrates that enum types are not allowed as range element types:

```
enum E { A = 0, B = 1, C = 2 }
constant r = A..C # Error: range E is not a a valid type
```

- This code illustrates that enum values can appear in ranges, after conversion to integers:

```
enum E { A = 0, B = 1, C = 2 }
constant r = 0..C # OK; equivalent to 0..2, with type range U8
```

Set Types

A **set type** represents an unordered collection of values.

Syntax

`set` *type*

Semantics

type is called the **element type** of the set type. It represents the type of each element of the set. It must be a range element type.

Examples

- The type of the expression `set { 0..1 }` is `set U8`.
- The type of the expression `set { [ 0, 1 ], [ 1, 2 ] }` is `set [2] U8`.
- The type of the expression `{ x = 0.0, y = 0.0 }, { x = 1.0, y = 1.0 }` is `set { x : F64, y : F64 }`.

Set Element Types

A **set element type** is the element type of a set type. It is a numeric type, an enum type, a structure type whose members all have set element type, an array type whose element type is a set element type, or a named type that refers to a set element type.

Examples

- `U32` is a set element type.
- `set U32` is a valid set type but not a set element type.
- `set set U32` is not a valid set type, because `set 32` is not a set element type.
- `set string` is not a valid set type, because `string` is not a set element type.
- `set { x : U32 }` is a valid set type.
- `set { x : string }` is not a valid set type because `{ x : string }` is not a set element type.
- `set { x : set U32 }` is not a valid set type because `{ x : set U32 }` is not a set element type (and also not a valid type).
- This code illustrates that enum types are allowed as set element types:

```
enum E { A = 0, B = 1, C = 2 }
constant s = set { A, B } # OK; type is set E
```

Named Types

A **named type** is a qualified identifier that refers to a type definition via the rules for name scoping. For example, in the example given for enum types above, `t` is a named type. Here is another example:

```
module M { type t = U32 }
constant x : M.t = 3 # M.t is a named type
```

State Data Types

A **state data type** is the type of a member of a state data definition. A state data type is either (1) a named type referring to a type definition whose representation type is a structure type or (2) a named type referring to a type definition whose representation type is a state data type.

For example:

```
type t1 = U32
type t2 = { x : U32 }
type t3 = t1
```

- `U32` and `{ x : U32 }` are not state data types, because they are not named types.
- `t1` is not a state data type, because its representation type is not a structure type or a named type referring to a state data type.
- `t2` and `t3` are state data types.

Expressions

Integer Literals

An **integer literal expression** is one of the following:

- A sequence of decimal digits `[0-9]` denoting the decimal representation of a nonnegative integer.
- `0x` or `0X` followed by a sequence of hexadecimal digits `[0-9A-Fa-f]` denoting the hexadecimal representation of a nonnegative integer.

An integer literal value v must be representable as an unsigned integer using most 64 bits, i.e., it must lie in the range 0 <= v < 2^64.

Examples

```
constant a = 123
constant b = 0xFF
```

Floating-Point Literals

A **floating-point literal expression** is a C-style representation of an IEEE floating-point number.

Examples

```
constant epsilon = 1e-10
constant delta = 0.001
constant pi = 3.14159
constant avogadro = 6.02E23
```

Boolean Literals

A **Boolean literal expression** is one of the values `true` and `false`.

Examples

```
constant a = true
constant b = false
```

String Literals

A **string literal expression** is a sequence of printable ASCII characters enclosed in double quotation marks. Each character in the expression represents a character of the string, except for the following character sequences:

- The sequence `\"` represents a quotation mark character.
- The sequence `\\` represents a backslash character.

Examples

- The expression `"abc"` represents the character sequence `a`, `b`, `c`.
- The expression `"\"abc\""` represents the character sequence `"`, `a`, `b`, `c`, `"`.
- The expression `"\\abc\\"` represents the character sequence `\`, `a`, `b`, `c`, `\`.
- The expression `"\abc\\"` also represents the character sequence `\`, `a`, `b`, `c`, `\`, because the sequence `\a` has no special meaning and is translated to the characters `\`, `a`.
- The expression `"\abc\"` is unterminated (and therefore invalid) because the final two characters `\"` translate to the single character `"` in the sequence, and so there is no terminating double quotation mark.

Arithmetic Expressions

tnet currently includes the following **arithmetic expressions**, with the usual precedence rules:

Syntax	Meaning
`-` e	Negate e
e_1 `+` e_2	Add e_1 and e_2
e_1 `-` e_2	Subtract e_2 from e_1
e_1 `*` e_2	Multiply e_1 by e_2
e_1 `/` e_2	Divide e_1 by e_2

For unary negation expressions, `-` e must have a type T according to the rules for type checking. If T is a structure type, then the result is computed by negating each member of e. If T is an array type, then the result is computed negating each element of e.

For binary arithmetic expressions, e_1 op e_2 must have a type T according to the rules for type checking. If T is a structure type, then the result is computed by applying op to the corresponding pairs of members of e_1 and e_2. If T is an array type, then the result is computed by applying op to the corresponding pairs of elements of e_1 and e_2.

Floating-point arithmetic occurs according to the rules for IEEE floating-point numbers. Integer arithmetic occurs according to the following rules:

1. Carry out the arithmetic using a bit width that is wide enough to hold the mathematical result. Division means the following:
   a. If a is zero and b != 0, then a / b evaluates to zero.
   b. Otherwise if b = 0, then throw an error.
   c. Otherwise if a and b have the same sign, then a / b evaluates to q, where q is the largest nonnegative integer such that q|b| <= |a|.
   d. Otherwise a / b evaluates to the arithmetic negation of (-a) / b.
2. If necessary, drop the high-order bits of the result to fit the width of the expression.

Because of the typing rules for expressions, dropping bits is required only where the mathematical result is wider than 64 bits.

Logical Expressions

A **logical expression** converts a pair of Boolean values into a Boolean value. Currently, the only logical expression available is `and`.

Syntax	Meaning
e_1 `and` e_2	Compute the logical and of e_1 and e_2

Membership Expressions

A **membership expression** tests for membership in a set.

Syntax

expression `in` *expression*

Semantics

The first expression must have type *T*, and the second expression must have type `set` *T*, after type conversion if necessary. The expression evaluates to `true` if the value represented by the first expression is a member of the set represented by the second expression, otherwise `false`.

Examples

```
1 in 1            # Evaluates to true
1 in 0..1         # Evaluates to true
1 in set { 0, 1 } # Evaluates to true
1 in set { 0, 2 } # Evaluates to false
```

Equality Expressions

An **equality expression** tests two subexpressions for equality.

Syntax

expression `=` *expression*

Semantics

The two subexpressions must have a common type *T*. The expression evaluates to `true` if the value represented by the first subexpression equals the value represented by the second subexpression, otherwise false.

Examples

```
1 = 1 # Evaluates to true
1 = 2 # Evaluates to false
```

Approximation Expressions

An **approximation expression** converts a value *v* into a range of values approximating *v*.

Syntax

expression `+-` *expression*

Semantics

The two subexpressions must have a common type *T*. *T* must be a numeric type. The expression e_1 `+-` e_2 evaluates to the range `(` e_1 `-` e_2 `)` `..` `(` e_1 `+` e_2 `)`.

Examples

```
1 +- 0.1 # Evaluates to 0.9..1.1
```

Structure Expressions

A **structure expression** represents a value of structure type, i.e., a data structure with members. Each member has a name and a value. The order of the members appearing in the value is not significant. Two or more members with the same name are not allowed.

Syntax

`{` *structure-member-sequence* `}`

structure-member-sequence is an element sequence in which each element is a structure member of the form *identifier* `=` *expression*, and the terminating punctuation is a comma. *identifier* is the name

of the member, and *expression* evaluates to its value.

Example

`{ x = 0, y = 1 }` is a structure expression with two members, `x` with value 0 and `y` with value 1. We can also write this value as follows:

```
{
  x = 0
  y = 1
}
```

Array Expressions

An **array expression** represents a value of array type, i.e., an indexed collection of values.

Syntax

`[` *array-element-sequence* `]`

array-element-sequence is an element sequence in which each element is an expression, and the terminating punctuation is a comma.

Examples

```
constant a = [ 0, 1, 2 ] # a has type [3] U8
constant b : [3] U32 = [
  3
  4
  5
]
```

Identifier Expressions

An **identifier expression** is an identifier that refers to a constant definition, enumerated constant definition, or state data definition. Its meaning is given by the rules for resolving identifiers.

Example

```
constant a = 42
constant b = a # a is an identifier expression
```

Dot Expressions

A **dot expression** represents one of the following:

1. A use that refers to a constant definition, enumerated constant definition, or state data

definition.

2. A member of a structure value.
3. The range value or set value formed by selecting a member of each element of an range or set of structure values.

Syntax

expression `.` *identifier*

Semantics

The following rules give the meaning of a dot expression e`.x`:

1. If e`.x` is a qualified identifier that represents one of the uses listed above according to the rules for resolving qualified identifiers, then it evaluates to the value stored in the corresponding definitions.
2. Otherwise if e evaluates to a structure value with a member `x`, then e`.x` evaluates to the value stored in the member.
3. Otherwise if e evaluates to a range value v_1 `..` v_2 , then e `.x` evaluates to the range value v_1 `.x..` v_2 `.x` if its elements are valid.
4. Otherwise if e evaluates to a set value `set {` v_1 `,` ... `,` v_n `}`, then e `.x` evaluates to the set value `set {` v_1 `.x,` ... `,` v_n `.x }` if its elements are valid.
5. Otherwise e`.x` is invalid.

Examples

Example 1

```
constant a = { x = 0, y = 1 }
constant b = a.y # a.y evaluates to 1
```

Example 2

```
constant a = { x = 1, y = 2 }..{ x = 3, y = 4 }
constant b = a.x # Evaluates to 1..3
```

Example 3

```
constant a = set { { x = 1, y = 2 }, { x = 3, y = 4 } }
constant b = a.x # Evaluates to set { 1, 3 }
```

Array Index Expressions

An **array index expression** represents one of the following:

- The value stored at a specified index of an array value.
- The range value or set value formed by selecting the value at a specified index of each element of a range or set of array values.

Syntax

expression [*expression*]

Semantics

The following rules give the meaning of an array index expression e_1 [e_2], where e_2 evaluates to the numeric value i, interpreted as a U64 value:

1. If e_1 evaluates to an array value v, and i is a valid index for v, then e_1 [e_2] evaluates to the value stored at index i of v.
2. Otherwise if e_1 evaluates to a range value v_1 .. v_2, then e_1 [e_2] evaluates to the range value v_1 [i].. v_2 [i] if its elements are valid.
3. Otherwise if e_1 evaluates to a set value set { v_1 , ... , v_n }, then e_1 [e_2] evaluates to the set value set { v_1 [i], ... , v_n [i] } if its elements are valid.
4. Otherwise e_1[e_2] is invalid.

Examples

Example 1

```
constant a = [ 1, 2, 3 ]
constant b = a[1] # a[1] evaluates to 2
```

Example 2

```
constant a = [ 1, 2 ]..[ 3, 4 ]
constant b = a[0] # Evaluates to 1..3
```

Example 3

```
constant a = set { [ 1, 2 ], [ 3, 4 ] }
constant b = a[0] # Evaluates to set { 1, 3 }
```

Parenthesis Expressions

A **parenthesis expression** is the usual construct for grouping subexpressions and enforcing evaluation order.

Syntax

`(` *expression* `)`

Semantics

Evaluate `expression` with the indicated evaluation order.

Example

```
constant c = (1 + 2) * 3
```

The parenthesis expression forces the evaluation of `+` before the evaluation of `*`.

Explicitly-Typed Expressions

An **explicitly-typed expression** converts a subexpression to a type.

Syntax

expression `:` *type*

Semantics

1. Evaluate *expression* to a value v.
2. Convert v to a value of type *type*. The type of *expression* must be convertible to *type*.

Example

```
constant x = 1 : F32
```

The expression on the right-hand side of the constant definition converts the integer literal 1 with type `U8` to type `F32`.

Range Expressions

A **range expression** represents a value of range type, i.e., all values of a range element type that lie between a lower bound and an upper bound.

Syntax

expression `..` *expression*

Semantics

Let e be the range expression e_1 `..` e_2, where e_1 and e_2 are expressions of type T_1 and T_2. Then T_1 and T_2 must be convertible to a common type T, and T must be a range element type. The expression e has type `range` T. It is evaluated as follows, where e_1 evaluates to v_1 and e_2

evaluates to v_2:

- If T is a numeric type, then e represents all values v of type T such that v_1 <= v <= v_2.
- If T is a structure type `{` x_1 `:` T_1 `,` ... `,` x_n `:` T_n `}`, then e represents all structure values `{` x_1 `=` w_1 `,` ... `,` x_n `=` w_n `}` such that for all i \in [1,n], w_i is an element of the range v_1.x_i `..` v_2.x_i.
- If T is an array type `[` e' `]` T, and e; evaluates to n after conversion to `U64`, then e represents all array values `[` w_0 `,` ... `,` w_(n-1) `]` such that for all i \in [0, n-1], w_i is an element of the range v_1 `[` i `]..` v_2 `[` i `]`.
- If T is an enum type, then e represents all values v of type T such that v_1 <= v <= v_2, where the ordering on the values of type T is given by the numeric values associated with the enumeration constants.
- If T is a named type with representation type T', then convert v_1 and v_2 to values v'_1 and v'_2 of type T' and apply these rules to the range v'_1 `..` v'_2.

Examples

```
constant a = 0..1
constant b = [ 0, 1 ]..[ 1, 2 ]
constant c = [ 0.0, 1.0 ]..[ 1.0, 2.0 ]
constant d = { x = 0, y = 0 }..{ x = 1, y = 1 }
constant e = { x = 0.0, y = 0.0 }..{ x = 1.0, y = 1.0 }
enum E { A = 0, B = 1, C = 2, D = 3 }
constant f = E.A..E.C
```

The constant `a` has type `range U8`. The values within this range are `0` and `1`.

The constant `b` has type `range [2] U8`. This range represents the values `[ 0, 1 ]`, `[ 0, 2 ]`, `[ 1, 1 ]`, and `[ 1, 2 ]`.

The constant `c` has type `range [2] F64`. This range represents all values `[` c_1 `,` c_2 `]` such that 0.0 <= c_1 <= 1.0 and 1.0 <= c_2 <= 2.0.

The constant `d` has type `range { x:U8, y:U8 }`. This range represents the values `{x=0, y=0}`, `{x=0, y=1}`, `{x=1, y=0}`, and `{x=1, y=1}`.

The constant `e` has type `range { x:F64, y:F64 }`. This range represents all values `{ x =` c_1 `, y =` c_2 `}` such that 0.0 <= c_1 <= 1.0 and 0.0 <= c_2 <= 1.0.

The constant `f` has type `range E` and represents the values `E.A`, `E.B`, and `E.C`.

Set Expressions

A **set expression** represents a value of set type, i.e., a set of point values and ranges of set element type.

Syntax

`set {` *set-element-sequence* `}`

set-element-sequence is an element sequence in which each element is an expression, and the terminating punctuation is a comma.

Semantics

Each expression in the set element sequence must evaluate to a single value of set element type or a range of values of set element type. The overall expression represents the union of all the elements represented by the single values and ranges.

Examples

- `set { 0..3, 5, 10 }` represents the following mathematical set of U8 values: \{ 0, 1, 2, 3, 5, 10 \}.
- `set { 0.0..1.0, 10.0 }` represents the following mathematical set of `F64` values: \{ x : 0.0 <= x <= 1.0 \} uu \{ 10.0 \}.

Precedence and Associativity

Precedence

Ambiguity in parsing expressions is resolved with the following precedence ordering. Expressions appearing earlier in the ordering have higher precedence. For example, `1 + 2 * 3` is parsed as `1 + (2 * 3)` and not `(1 + 2) * 3`.

- Dot expressions *e* `.` *e*.
- Array index expressions *e* `[` *e* `]`.
- Unary minus expressions `-` *e*.
- Range expressions *e* `..` *e*.
- Explicitly typed expressions *e* `:` *t*.
- Multiplication expressions *e* `*` *e* and division expressions *e* `/` *e*.
- Addition expressions *e* `+` *e* and subtraction expressions *e* `-` *e*.
- Approximation expressions *e* `+-` *e*.
- Membership expressions *e* `in` *e*.
- Equality expressions *e* `=` *e*.
- Logical expressions *e* `and` *e*.

Associativity

Wherever necessary to resolve ambiguity, operators are left associative. For example, `a = b = c` is parsed as `(a = b) = c` and not `a = (b = c)`.

Definitions

The tnet language currently supports the following kinds of definitions:

- Type Definitions
- Constant Definitions
- Enum Definitions
- Enumerated Constant Definitions
- State Data Definitions
- Timeline Definitions
- Command Definitions
- Task Template Definitions
- Task Network Definitions
- Module Definitions

Type definitions, constant definitions, enum definitions, state data definitions, timeline definitions, command definitions, task template definitions, task network definitions, and module definitions may appear in translation units or in module definitions. Enumerated constant definitions appear in enum definitions.

Type Definitions

A **type definition** defines a new type and gives it a name. The new type is associated with an existing type, called the **representation type**. Elsewhere in the program, you can use the defined type.

Syntax

`type` *identifier* `=` *type*

Semantics

The identifier on the left-hand side is the name of the defined type. The type on the right-hand side of the definition is the representation type. We say that the named type **represents** its representation type.

At any program point where the qualified identifier Q refers to a definition `type` I `=` T according to the scoping rules for names, you can use Q.

If the representation type is a structure type, then the representation type may not have a structure or array type for any of its members. Named types that represent structure or array types are allowed as the member types.

If the representation type is an array type, then the representation type may not have a structure or array type for its element type. A named type that represents a structure or array type is allowed as the element type.

Examples

```
type VectorF32 = { x : F32, y : F32, z : F32 }
constant v = { x = 0, y = 0, z = 0 } : VectorF32
```

```
enum PorridgeStatus { NOT_EATEN = 0, EATEN = 1 }
type GoldilocksStatus = {
  mama : PorridgeStatus
  papa : PorridgeStatus
  baby : PorridgeStatus
}
constant status : GoldilocksStatus = {
  mama = PorridgeStatus.EATEN
  papa = PorridgeStatus.NOT_EATEN
  baby = PorridgeStatus.NOT_EATEN
}
```

In the second example, the type `GoldilocksStatus` is defined with type a definition. `PorridgeStatus` is an enum type defined with an enum definition.

This example is not allowed, because the representation type is a structure type with an unnamed structure type in one of its members:

```
type t = { x : { y : U32 } } # Error
```

This example is allowed:

```
type t1 = U32 # OK
type t2 = { x : t1 } # OK
```

Constant Definitions

A **constant definition** associates a name with a compile-time constant value. You can use the name in place of the value elsewhere in the program.

Syntax

`constant` *identifier* *[* `:` *type* *]* `=` *expression*

Semantics

type is optional. If it is present, then the type of *expression* must be convertible to *type*.

If *type* is present, then the type of the constant is *type*. Otherwise, the type of the constant is the type of *expression*.

expression must evaluate to a compile-time constant value *v*. At any program point where the qualified identifier *Q* refers to the constant definition according to the scoping rules for names, you can use *Q* as a name for the value that results from converting *v* to the type of the constant.

Examples

```
constant a = 0 # a has value 0 : U8
constant b : U32 = 1 # b has value 1 : U32
constant c = 2 : F32 # c has value 2.0 : F32
constant d : { x : F32, y : F32 } = { x = 0, y = 1 } # d has value { x = 0, y = 1 } :
{ x : F32, y : F32 }
```

Enum Definitions

An **enum definition** does two things:

1. It defines a type T and associates a name N with T. Elsewhere in the program, you can use N to refer to T.
2. It associates several named constants C_1, ..., C_n with T. These constants, called the **enumerated constants** of T, are the values that an expression of type T may attain. Elsewhere in the program, you can use the qualified identifiers N . C_1, ..., N . C_n to refer to the enumerated constants.

Syntax

`enum` *identifier* *[* `:` *type* *]* `{` *enum-constant-sequence* `}`

enum-constant-sequence is an element sequence in which the elements are enumerated constant definitions, and the terminating punctuation is a comma.

Semantics

The enumerated constants have the enum type defined in the enum definition. In translation, they are represented as values of integer type, called the **representation type**.

Inferred Representation Type

The semantic analyzer infers a representation type from the types of the expressions e_1, ... , e_n in the enumerated constants as follows:

1. Check that n > 0. If not, then throw an error.
2. For each i in [1, n]
   a. Compute the type T_i.
   b. Check that T_i is an integer type. If not, throw an error.
3. Compute the common type T of the list T_1, ..., T_n.
4. Use the type T as the representation type.

Optional Explicit Representation Type

If the optional type T appears after the identifier, then the semantic analyzer does the following:

1. Check that T is an integer type. If not, throw an error.
2. Use T as the representation type.

Example

```
enum U8Gunfighters {
  IL_BUONO = 0
  IL_MALO = 1
  IL_CATTIVO = 2
}

enum U32Gunfighters : U32 {
  IL_BUONO = 0
  IL_MALO = 1
  IL_CATTIVO = 2
}

enum DifferentSizes {
  A = 0
  B = 1
  C = 256
}
```

Here are three enum definitions. In the first one, the implicit representation type is `U8`, because all the enumerated constants are of type `U8`. In the second one, the representation type is explicitly given as `U32`. In the third one, the representation type is `U16`, because the largest constant value (256) is a `U16`.

Enumerated Constant Definitions

An **enumerated constant definition** is an element of an enum definition. Like a constant definition, it associates a value with a named constant. It also establishes that the constant is one of the values of the type defined in the enum definition.

Syntax

identifier `=` *expression*

Semantics

expression must evaluate to a compile-time constant value *v*. At any program point where the qualified identifier *Q* refers to the enumerated constant definition according to the scoping rules for names, you can use *Q* as a name for the value that results from converting *v* to the type of the enclosing enum definition.

Note that the type of an enumerated constant value is the enum type defined by the enclosing enum definition. This may be converted to the representation type of the enum. However, the reverse conversion is not allowed: you can convert an enum type to a `U32` (for example), but not a `U32` to an enum type. Nor can you convert one enum type to a different one.

Example

The example given for enum definitions includes enumerated constant definitions.

State Data Definitions

A **state data definition** does the following:

1. Define a kind of data stored in the state database.
2. Associate a name with the kind of data.
3. Specify (as a structure type) the type of the data.
4. Specify the initial value of the data.

You can refer to members of the structure (or members of their members) in planning timelines.

Syntax

`state` *identifier* *[* `:` *type* *]* `=` *expression*

Semantics

- The type to the right of the identifier is optional. If it appears, it must be a state data type.
- If a type T appears to the right of the identifier, then the type of the expression to the right of the equals sign must be convertible to T.
- Otherwise the type of the expression to the right of the equals sign must be a state data type.

Examples

See the examples in the Overview section on defining state data.

Timeline Definitions

A **timeline definition** identifies an element of a state data structure as a variable stored in a planning timeline.

Syntax

`timeline` *expression* *[* `in` *expression* *]*

Semantics

- The first expression must be a dot expression or an array index expression, and it must refer to a **transitive element** of a structure S appearing in a state data definition. A transitive element of S is one of the following: (1) a structure member of S; (2) a structure member of a transitive

element of S; or (3) an array cell of a transitive element of S. For example, if `s` refers to a state data definition S, then `s.x`, `s.x.y`, and `s.x[0]` are all transitive elements of S.

- Type checking the first expression must produce a valid type T, and T must be a set element type.
- The part of a timeline definition starting with the keyword `in` is called the **in clause**. It is optional. If the in clause appears, then type checking the expression in the in clause must produce a type that is convertible to the set type `set` T.
- If no in clause appears, then the defined timeline may attain all the values that are legal for the type T. Otherwise, the timeline is restricted to the values specified by the expression in the in clause, after evaluating the expression and converting the value to a set value if necessary.

Examples

See the examples in the Overview section on timeline definitions.

Command Definitions

A **command definition** defines a spacecraft command that may be issued within a task body.

Syntax

`command` *identifier* *[* `(` *parameter-list* `)` *]* `opcode` *expression*

Semantics

The identifier names the command.

The parameter list provides the command parameters. If there are no parameters, then the parameter list may be omitted. Each type appearing in a parameter must be a **command parameter type**. A command parameter type is a primitive type, a string type, an enum type, or a named type that refers to a command parameter type.

The expression must have numeric type and must be a compile-time constant. It provides the opcode for the command.

Task Template Definitions

A **task template definition** defines a task template. A task template specifies a task either partially or completely. It may be instantiated into a task (possibly with further specification) as part of a task network definition.

Syntax

A task template is one of the following:

- A **with task template** `task template` *identifier* *[* `(` *parameter-list* `)` *]* `with` `{` *task-body* `}`
- A **from task template** `task template` *identifier* *[* `(` *parameter-list* `)` *]* `from` *expression* *[* `(` *argument-list* `)` *]* *[* `with` `{` *task-body* `}` *]*

Semantics

With task templates: The identifier is the name of the task template. The parameters are the parameters of the task body. Each parameter must have set element type. If there are no parameters, then the parameter list may be omitted. The task body specifies the elements of the task template.

From task templates: The identifier is the name of the task template. The parameters are the parameters of the task body. Each parameter must have set element type. If there are no parameters, then the parameter list may be omitted.

The expression following the keyword `from` must refer to another task template *T*. The arguments are the bindings for the parameters of *T*. If there are no arguments, then the argument list may be omitted. The number and type of arguments must match the number and type of parameters of *T*.

The task body following the keyword `with` is optional. If a task body *B* is present, then the elements of the template are formed as follows:

1. Take the union of the elements of *T* and the elements given in *B*.
2. If any kind of element appears both in *B* and in *T*, and only one such element is allowed in a task body, then the element appearing in *B* overrides the element appearing in *T*.

Parameter Lists

A **parameter list** is a list of typed parameters.

Syntax

A parameter list is an element sequence in which each element is a parameter, and the terminating punctuation is a comma.

A parameter has the following syntax:

identifier `:` *type*

Semantics

Each parameter defines a named, typed constant value that may be used by name in the scope where the parameters are defined. The identifier names the parameter. The same identifier may not appear twice in the list.

Argument Lists

A **argument list** is a list of arguments.

Syntax

An argument list is an element sequence in which each element is an expression, and the terminating punctuation is a comma.

Semantics

An argument list provides the bindings to a parameter list.

Task Bodies

A task body is the body of a task template or task.

Syntax

A **task body** is an element sequence in which the elements are task body elements, and the terminating punctuation is a semicolon.

A **task body element** is one of the following:

- A **start element** `start` *expression*
- A **start range element** `start in` *expression*
- A **duration element** `duration` *expression*
- An **end range element** `end in` *expression*
- A **precondition element** `pre` *expression*
- A **maintenance condition element** `maint` *expression*
- A **postcondition element** `post` *expression*
- An **impact element** `impact` *expression* `=` *expression*
- A **cumulative impact element** `impact` *expression* `+=` *expression*
- A **rate impact element** `rate impact` *expression* `+=` *expression*
- A **command element** `command` *expression* *[* `(` *argument-list* `)` *]*
- A **planning command element** `planning command` *expression* *[* `(` *argument-list* `)` *]*
- A **cleanup command** element `cleanup command` *expression* *[* `(` *argument-list* `)` *]*
- A **with contingency element** `contingency with {` *task-body* `}`
- A **from contingency element** `contingency from` *expression* *[* `(` *argument-list* `)` *]* *[* `with {` *task-body* `}` *]*

Semantics

Start elements: A task body may contain at most one start element. The expression must have a numeric type. It denotes the start time of the task in seconds of absolute spacecraft time.

Start range elements: A task body may contain at most one start range element. The expression must be of type `range` *T*, where *T* is a numeric type. It denotes the bounds on the start time of the task, in seconds of absolute spacecraft time.

Duration elements: A task body may contain at most one duration element. The expression must be of numeric type. It denotes a task duration in seconds.

End range elements: A task body may contain at most one end range element. The expression

must be of type `range` *T*, where *T* is a numeric type. It denotes the bounds on the end time of the task, in seconds of absolute spacecraft time.

Precondition elements: A task body may at most one precondition element. The expression must have Boolean type. It denotes a condition that must be true before the task is executed.

Maintenance condition elements: A task body may contain at most one maintenance condition element. The expression must have Boolean type. It denotes a condition that must be true while the task is executed.

Postcondition elements: A task body may contain at most one postcondition element. The expression must have Boolean type. It denotes a condition that must be true after the task is executed.

Impact elements: A task body may contain zero or more impact elements. The first expression must refer to a timeline. The second expression must have the same type as the timeline. The element denotes the impact, or update, that results from evaluating the second expression to a value *v* and storing storing *v* into the timeline.

Cumulative impact elements: A task body may contain zero or more cumulative impact elements. The first expression must refer to a timeline of numeric type. The second expression must have the same type as the timeline. The element denotes the impact, or update, that results from (1) evaluating the second expression to a value *v*, (2) adding *v* to the value already in the timeline to produce a new value v', and (3) storing v' into the timeline.

Rate impact elements: A task body may contain zero or more rate impact elements. The first expression must refer to a timeline of numeric type. The second expression must have the same type as the timeline. The element denotes a change to the rate associated with the timeline. The new rate is computed by (1) evaluating the second expression to a value *v*, (2) adding *v* to the old rate.

Command elements: A task body may contain at most one command element. The expression must refer to a command definition. The command is issued when the task is executed. The number and types of the arguments must match the command parameters. If there are no arguments, then the argument list may be omitted.

Planning command elements: A task body may contain at most one planning element. The expression must refer to a command definition. The command is issued when the task is planned, for example to command the flight software to send additional information to the planner. The number and types of the arguments must match the command parameters. If there are no arguments, then the argument list may be omitted.

Cleanup command elements: A task body may contain at most one cleanup element. The expression must refer to refer to a command definition. The command is issued when the task has been dispatched, and its execution has failed. The number and types of the arguments must match the command parameters. If there are no arguments, then the argument list may be omitted.

With contingency elements: A task body may contain zero or more with contingency elements. Each with contingency element denotes an anonymous task template whose elements are given by the task body. The template is instantiated to a task and added to the plan if and when the task

execution fails.

From contingency elements: A task body may contain zero or more from contingency elements. In each from contingency element, the expression must refer to a task template definition *T*. The arguments are the bindings for the parameters of *T*. If there are no arguments, then the argument list may be omitted. The number and type of arguments must match the number and type of parameters of *T*.

The task body *B* following the keyword `with` is optional. If it is present, then the from contingency element denotes a task template whose elements are given by adding the task body to the template, as for task template definitions.

The template is instantiated to a task and added to the plan if and when the task execution fails.

Task Network Definitions

A **task network definition** defines a task network.

Syntax

`tasknet` *identifier* { *task-network-body* }

Semantics

The identifier names the task network. The task network body specifies the timelines and tasks of the task network.

Task Network Bodies

A **task network body** is the body of a task network. A task network body consists of zero or more timelines, zero or more task templates, and zero or more tasks.

Syntax

task-network-body is an element sequence in which the elements are task network body elements, and the terminating punctuation is a semicolon.

A **task network body element** is one of the following:

- A **timeline element** `timeline` *expression*
- A **task template element** `task template` *expression* *[* `(` *argument-list* `)` *]* *[* `with` { *task-body* } *]*
- A **with task element** `task` *identifier* `with` { *task-body* }
- A **from task element** `task` *identifier* `from` *expression* *[* `(` *argument-list* `)` *]* *[* `with` { *task-body* } *]*

Semantics

Timeline elements: The expression must refer to a transitive element E of a state data structure, as for a timeline definition. E must also appear in a timeline definition that is in scope. The timeline element imports the timeline associated with E into the task network, so that when the task

network is loaded into the planner, the timeline is created.

Task template elements: The expression must refer to a task template definition. The arguments are default bindings for the parameters of *T*. If there are no arguments, then the argument list may be omitted. The number and type of arguments must match the number and type of parameters of *T*.

The element imports the task template into the task network, so that when the task network is loaded into the planner, the template is created. During planning the planner can instantiate the template, replacing one or more of the default parameter bindings with new values if necessary.

The task body *B* following the keyword `with` is optional. If it is present, then the element creates a template whose elements are given by adding *B* to the template, as for task template definitions.

With task elements: The identifier is the name of the task. The element creates a template with the elements given in the task body and adds it as task with the given name.

From task elements: The identifier is the name of the task. The expression must refer to a task template definition *T*. The arguments are the bindings for the parameters of *T*. If there are no arguments, then the argument list may be omitted. The number and type of arguments must match the number and type of parameters of *T*.

The task body *B* following the keyword `with` is optional. If it is present, then the element creates a template whose elements are given by adding *B* to the template, as for task template definitions.

The element instantiates the template and adds it as a task with the given name.

Module Definitions

A **module definition** provides a named scope that encloses other definitions, including other module definitions.

Syntax

`module` *identifier* `{` *module-member-sequence* `}`

module-member-sequence is an element sequence in which each element is any kind of definition except an enumerated constant definition, and the terminating punctuation is a semicolon. Syntactically, it is equivalent to a translation unit.

Semantics

A module definition *D* qualifies the names of all the definitions inside it with its own name. Inside *D*, you can refer to definitions in *D* by their unqualified name (the identifier appearing in the definition) or by their qualified name. Outside *D*, you have to use the qualified name. We say that the **scope** of the identifiers in the definitions in *D* is limited to the inside of *D*.

For further information about name scoping and qualification, see the section on Scoping of Names.

Example

```
module M {
  constant a = 0
  constant b = a # Inside M, we can refer to M.a as a
  constant c = M.a # We can also say M.a here
}
constant d = M.a # Outside M, we have to say M.a
constant e = a # Error: a is not in scope here
```

Element Sequences

An **element sequence** is a sequence of similar elements, e.g., the definitions appearing in a translation unit, the definitions enclosed in a module definition, or the elements of an array expression.

Each element of an element sequence has optional **terminating punctuation**. The punctuation is either a comma or a semicolon, depending on the kind of sequence.

For each element *e* in the sequence:

- You can always terminate *e* with a line break. In this case no terminating punctuation is required.
- You can omit the line break. In this case the terminating punctuation is required, unless the element is last in the sequence.

Examples

A translation unit is a kind of element sequence. Here are some examples of element sequences using constant definitions as the elements of a translation unit:

```
# No terminating punctuation
constant a = 0
constant b = 1
```

```
# Optional terminating punctuation present
constant a = 0;
constant b = 1;
```

```
# Terminating punctuation required after the first element but not the second
constant a = 0; constant b = 1
```

```
# Error, because terminating punctuation is missing
constant a = 0 constant b = 1
```

An enum constant sequence is another example of an element sequence. Here are some element sequences using enumerated constant definitions as elements of an enum constant sequence:

```
# No terminating punctuation
enum E {
  X = 0
  Y = 1
}
```

```
# Optional terminating punctuation
enum E {
  X = 0,
  Y = 1,
}
```

```
# Terminating punctuation required after the first element but not the second
enum E { X = 0, Y = 1 }
```

Comments and Annotations

Comments

A **comment** is program text that is ignored by the translator. It provides information to human readers of the source program.

Comments begin with the character # and go to the end of the line. For example:

```
# This is a comment
```

To write a multiline comment, precede each line with #:

```
# This is a multiline comment.
# It has two lines.
```

Annotations

An **annotation** is similar to a comment, but it is attached to a syntactic element of a program, and it is preserved during translation. The precise use of annotations depends on the translator. A typical use is to include annotations as comments in generated code.

Where Annotations Can Occur

Annotations can occur at the following syntax elements:

- A definition.
- A parameter.
- A member of a structure type.

These elements are called the **annotatable elements** of a tnet program.

Kinds of Annotations

There are two kinds of annotations: **pre-annotations** and **post-annotations**.

A pre-annotation starts with `@` and goes to the end of the line. Whitespace characters after the initial `@` are ignored. A pre-annotation must occur immediately before an annotatable element *e*. It is attached to *e* during translation.

A post-annotation starts with `@<` and goes to the end of the line. Whitespace characters after the initial `@<` are ignored. A post-annotation must occur immediately after an annotatable element *e*. It is attached to *e* during translation.

Example

```
@ This is module M
module M {
  constant a = 0 @< This is constant M.a
  constant b = 1 @< This is constant M.b
  @ This is an enum
  enum E {
    a = 0 @< This is enumerated constant M.E.a
    b = 1 @< This is enumerated constant M.E.b
  }
}
```

Multiline Annotations

You can write several pre-annotations in a row before an annotatable element *e*. In this case, all the pre-annotations are attached to the element, in the order that they appear.

Similarly, you can write several post-annotations in a row after an annotatable element *e*. In this case, all the post-annotations are attached to the element, in the order that they appear.

Example

```
@ This is module M
@ Its pre-annotation has two lines
module M {
  constant a = 0 @< This is constant M.a
                 @< Its post-annotation has two lines
  constant b = 1 @< This is constant M.b
                 @< Its post-annotation has two lines
  @ This is an enum
  @ Its pre-annotation has two lines
  enum E {
    a = 0 @< This is enumerated constant M.E.a
          @< Its post-annotation has two lines
    b = 1 @< This is enumerated constant M.E.b
          @< Its post-annotation has two lines
  }
}
```

Translation Units and Programs

Translation Units

A **translation unit** forms part of a program at the top level.

Syntax

A translation unit is an element sequence in which each element is any kind of definition except an enumerated constant definition, and the terminating punctuation is a semicolon. Syntactically, it is identical to a module member sequence.

Example

Here is a translation unit:

```
module M1 { constant a = 0 }
```

And here is another one:

```
module M1 { constant b = 0 }
```

And here is a third one:

```
module M2 { constant a = M1.a + M1.b }
```

Call these translation units 1, 2, and 3 for purposes of the example in the following section.

Programs

A **program** is a collection of one or more translation units. A program is presented to one or more translators for translation. How this is done depends on the translator. Typically, you ask a translator to read a single translation unit from standard input and/or to read one or more translation units stored in files, one unit per file.

Example

Translation units 1-3 in the previous section, taken together, form a single program. That program is equivalent to the following single translation unit:

```
module M1 { constant a = 0 }
module M1 { constant b = 1 }
module M2 { constant a = M1.a + M1.b }
```

According to the semantics of module definitions, this is also equivalent:

```
module M1 {
  constant a = 0
  constant b = 1
}
module M2 { constant a = M1.a + M1.b }
```

Note that translation unit 3 alone is not a valid program, because it uses free symbols defined in the other translation units. Similarly, the translation unit 3 together with just translation unit 1 or translation unit 2 is not a valid program.

Scoping of Names

Qualified Identifiers

A **qualified identifier** is one of the following:

1. An identifier.
2. *Q* . *I*, where *Q* is a qualified identifier and *I* is an identifier.

Examples:

```
a
a.b
a.b.c
```

Names of Definitions

Every definition *D* appearing in a tnet program has a unique **qualified name**. The qualified name is a qualified identifier formed as follows:

- If *D* appears outside any module definition or enum definition, then the qualified name is the identifier *I* appearing in *D*.
- Otherwise, the qualified name is *N* `.` *I*, where *N* is the qualified name of the enclosing module definition or enum definition, and *I* is the identifier appearing in *D*.

For example:

```
module M { # Qualified name is M
  module N { # Qualified name is M.N
    enum Maybe { # Qualified name is M.N.Maybe
      NO = 0 # Qualified name is M.N.Maybe.NO
      YES = 1 # Qualified name is M.N.Maybe.YES
    }
  }
}
```

Names of Parameters

Every parameter that is in scope has a unique name. The name is the identifier that appears in the parameter. It is always a simple identifier.

Namespaces

The qualified names of definitions and the names of parameters reside in **namespaces**. There are four namespaces: the **type namespace**, the **value namespace**, the **command namespace**, and the **task namespace**. The following table shows which names reside in which namespaces.

Kind of Name	Type Namespace	Value Namespace	Command Namespace	Task Namespace
Type definition	X			
Constant definition		X		
Enumerated constant definition		X		
Module definition		X		
Enum definition	X	X		
Parameter		X		
Command definition			X	

Kind of Name	Type Namespace	Value Namespace	Command Namespace	Task Namespace
Task template definition				X
Task definition				X

An X means that the name resides in the namespace corresponding to the column. Note that enum definitions reside in both the type namespace and the value namespace.

Multiple Definitions with the Same Qualified Name

Different Namespaces

Two definitions with the same qualified name are allowed if they are in different namespaces. For example:

```
type t = U32 # Defines t in the type namespace
constant t : t = 0 # Defines t in the value namespace
```

Module Definitions

Multiple syntactic module definitions with the same qualified name are allowed. The semantic analysis combines all such definitions into a single module definition with that qualified name. For example, this program is legal

```
module M { constant a = 0 }
module M { constant b = 1 }
constant c = M.a + M.b
```

It is equivalent to this program:

```
module M {
  constant a = 0
  constant b = 1
}
constant c = M.a + M.b
```

Because the order of definitions is irrelevant, this is also equivalent:

```
module M { constant a = 0 }
constant c = M.a + M.b
module M { constant b = 1 }
```

Conflicting Definitions

Within the same namespace, two definitions with the same qualified name are not allowed, unless they are both module definitions as described above. For example:

```
module M {
  constant a = 0
  constant a = 1 # Error: Name M.a is redefined
}
```

Two definitions with the same identifier are allowed if they have different qualified names, for example:

```
constant a = 0
module M {
  constant a = 1 # OK, qualified name is M.a =/= a
}
```

Resolution of Identifiers

The following rules govern the resolution of identifiers, i.e., associating identifiers with definitions:

1. Use the context to determine which namespace *S* to use. For example, if we are expecting a type name, then use the type namespace.
2. At the top level (outside of any module definition, enum definition, task template definition, or task definition), the identifier *I* refers to the unique definition with qualified name *I* if it exists in namespace *S*. Otherwise an error results.
3. Inside a module definition, enum definition, task template definition, or task definition with qualified name *N* appearing at the top level:
   a. If the identifier *I* names a parameter in scope, then it refers to the parameter.
   b. Otherwise the identifier *I* refers to the definition with qualified name *N* . *I* if it exists in namespace *S*.
   c. Otherwise *I* refers to the definition with qualified name *I* if it exists in namespace *S*.
   d. Otherwise an error results.
4. Inside a module definition, enum definition, task template definition, or task definition with qualified name *N* appearing inside a module definition *D*:
   a. If the identifier *I* names a parameter in scope, then it refers to the parameter.
   b. Otherwise the identifier *I* refers to the definition with qualified name *N* . *I* if it exists in namespace *S*.
   c. Otherwise proceed as if *I* were appearing inside *D*.

Example

S refers to the value namespace.

```
# Identifier M is in scope in S and refers to the qualified name M
# Identifier a is in scope in S and refers to qualified name a

constant a = 1 # Unique definition in S with qualified name a

module M {
  # Identifier M is in scope in S and refers to the qualified name M
  # Identifier N is in scope in S and refers to the qualified name N
  # Identifier a is in scope in S and refers to qualified name a
  # Identifier b is in scope in S and refers to qualified name M.b
  constant b = 2 # Unique definition in S with qualified name M.b
}

# Identifier M is in scope in S and refers to the qualified name M
# Identifier a is in scope in S and refers to qualified name a

module M {

  # Identifier M is in scope in S and refers to the qualified name M
  # Identifier N is in scope in S and refers to the qualified name M.N
  # Identifier a is in scope and refers to qualified name a
  # Identifier b is in scope and refers to qualified name M.b

  module N {
    # Identifier M is in scope in S and refers to the qualified name M
    # Identifier N is in scope in S and refers to the qualified name M.N
    # Identifier a is in scope in S and refers to qualified name a
    # Identifier b is in scope in S and refers to qualified name M.N.b
    constant b = 3 # Unique definition in S with qualified name M.N.b
  }

}

# Identifier M is in scope in S and refers to the qualified name M
# Identifier a is in scope in S and refers to qualified name a
```

Resolution of Qualified Identifiers

The following rules govern the resolution of qualified identifiers, i.e., associating qualified identifiers with definitions:

1. If a qualified identifier is an identifier, then resolve it as stated in the previous section.
2. Otherwise, the qualified identifier has the form *Q* . *I*, where *Q* is a qualified identifier and *I* is an identifier. Do the following:

a. Recursively resolve *Q*.

b. If *Q* refers to a module definition or enum definition *D*, then do the following:

   i. Determine the namespace *S* of *Q* `.` *I*.

   ii. Look in *D* for a definition with identifier *I* in namespace *S*. If there is none, issue an error.

c. Otherwise, if *Q* refers to a value of structure type, then look in the type of the value for a structure member with identifier *I*. If there is none, then issue an error.

d. Otherwise, if *Q* refers to a value of range or set type *T* whose element type *T′* is a structure type, then look in *T′* for a structure member with identifier *I*. If there is none, then issue an error.

e. Otherwise the qualified identifier is invalid. Issue an error.

The rules for name conflicts are designed so that rule 2(b)(i) makes sense. For example, the expression `a.b` can never refer to both a structure member and a type. From the definition associated with `a`, we can determine what the namespace of `a.b` must be.

Example

```
module M {
  constant a = { x = 0, y = 1 }
  enum E {
    b = 2
    c = b # Refers to M.E.b
    d = E.b # Refers to M.E.b
    e = M.E.b # Refers to M.E.b
  }
  constant f = a # Refers to M.a
  constant g = M.a # Refers to M.a
  constant h = E.b # Refers to M.E.b
  constant i = M.E.b # Refers to M.E.b
  constant j = a.x # Refers to structure member x of M.a
  constant k = M.a.x # Refers to structure member x of M.a
}
```

Definitions and Uses

Uses

A **use** is a qualified identifier appearing in an expression or a type that refers to a definition according to the scoping rules for names.

Not all qualified identifiers are uses. For example, in line 2 of this program

```
constant c1 = { x = 0 }
constant c2 = c1.x
```

`c1` is a use, but `c1.x` is not. `c1` refers to the definition of the constant `c1`, while `c1.x` refers to an element of a structure value.

Use-Def Graph

The set of definitions and uses in a tnet program induces a directed graph called the **use-def graph**. In this graph,

1. The nodes are the definitions.
2. There is an edge from each definition *d* to the definitions d_1, ..., d_n corresponding to the uses u_1, ..., u_n appearing in *d*.

For example, in the code above, the use-def graph has two nodes `c1` and `c2` and one edge `c2` rarr `c1`.

In a legal tnet program, the use-def graph must be acyclic. For example, this program is illegal:

```
constant a = b
constant b = a
```

This program is also illegal:

```
constant a = a
```

Order of Definitions and Uses

So long as the use-def graph is acyclic, there is no constraint either on the ordering of definitions and uses within a translation unit, or on the distribution of definitions and uses among translation units. For example, if the definition `constant c = 0` appears anywhere in any translation unit of a program *P*, then the use of `c` as a constant value of type U8 is legal anywhere in any translation unit of *P*. In particular, this program is legal:

```
constant b = a
constant a = 0
```

The program consisting of two translation units

```
constant b = a
```

and

```
constant a = 0
```

is also legal, and the order in which the units are presented to the translator does not matter.

Type Checking

In this section, we explain the rules used to assign types to expressions. tnet is a statically typed language. That means the following:

- Type checking of expressions occurs during translation.
- If the type checking phase detects a violation of any of these rules, then translation halts with an error message and does not produce any code.

Each type represents a collection of values. The type checking rules exist to ensure that whenever an expression of type T is evaluated, the result is a value of type T.

Integer Literals

To type an integer literal expression, the semantic analyzer does the following:

1. Evaluate the expression to an unsigned integer value v.
2. Check that v can be represented in 64 or fewer bits. If not, throw an error.
3. Use the narrowest unsigned integer type that can represent v.

Examples

```
constant a = 0 # Type is U8
constant b = 1 # Type is U8
constant c = 256 # Type is U16
constant d = 65536 # Type is U32
constant e = 0x100000000 # Type is U64
constant f = 0x10000000000000000 # Error; integer value is too large
```

Floating-Point Literals

The type of a floating-point literal expression is `F64`.

Boolean Literals

The type of a boolean literal expression is `boolean`.

String Literals

The type of a string literal expression is `string`.

Unary Minus

To type a unary minus expression `-` e, the semantic analyzer does the following:

1. Compute the type T of e.
2. If T is a named type, then replace the named type with its representation type and reapply these rules.
3. Otherwise if T is a structure type, then
   a. For each member m_i `:` T_i of T, apply these rules to compute the type T'_i of `-` e_i, where e_i has type T_i.
   b. Use the structure type with members m_i `:` T'_i.
4. Otherwise if T is an array type `[` n `]` T', then
   a. Apply these rules to compute the type T'' of `-` e, where e has type T'.
   b. Use the type `[` n `]` T''.
5. Otherwise
   a. Check that T is a primitive type T'. If not, throw an error.
   b. If T' is a floating point type or signed integer type, then use T'.
   c. Otherwise if T' is U64, then use I64.
   d. Otherwise use the narrowest signed integer type whose width is larger than the width of T'.

Examples

```
constant a = - { x = 1 } # Type is { x: I16 }
constant b = -1.0 # Type is F64
constant c = -1 # Type is I16
constant d = -c # Type is I16
constant e = -0xFFFFFFFF # Type is I64
constant f = -0x100000000 # Type is I64
```

Binary Arithmetic Expressions

To type a binary arithmetic expression e_1 *op* e_2, the semantic analyzer does the following:

1. Compute the type T_1 of e_1 and the type T_2 of *e_2*.
2. If either of T_1 or T_2 is a named type, then replace each named type with its representation type and reapply these rules.
3. Otherwise if T_1 and T_2 are both structure types that are convertible to a common type T, then

a. For each member name m_i appearing in T, apply these rules to compute the type T'_i of e_{1i} op e_{2i}, where e_{1i} has the type of member m_i of T_1, and e_{2i} has the type of member m_i of T_2.

b. Use the structure type with members m_i : T'_i.

4. Otherwise if T_1 and T_2 are both array types with size n that are convertible to a common type, then

   a. Apply these rules to compute the type T' of e_1 op e_2, where the type of e_1 is the member type of T_1, and the type of e_2 is the member type of T_2.

   b. Use the type [n] T'.

5. Otherwise if one of T_1 and T_2 is a primitive type and the other one is a structure type S, then

   a. Let \{ m_i \} be the set of member names in the structure type.

   b. Let T be the primitive type.

   c. Let T' be the structure type with members m_i : T.

   d. Replace the primitive type with T' and reapply these rules.

6. Otherwise if one of T_1 and T_2 is a primitive type T' and the other one is an array type [n] T'', then replace the primitive type with [n] T' and reapply these rules.

7. Otherwise

   a. Check that T_1 and T_2 are both primitive types. If not, throw an error.

   b. Resolve T_1 and T_2 to a common type T as follows:

      i. If T_1 and T_2 are both floating-point types, then use the floating-point type with the same width as the maximal width of T_1 and T_2.

      ii. Otherwise if one of T_1 and T_2 is an integer type and the other is a floating-point type, then use the floating-point type.

      iii. Otherwise if *op* is - and T_1 or T_2 has width 64, then use I64.

      iv. Otherwise if *op* is - then use the narrowest signed integer type whose width is larger than the maximal width of T_1 and T_2.

      v. Otherwise if *op* is / and T_2 is unsigned, then use T_1.

      vi. Otherwise if *op* is / and T_1 has width 64, then use I64.

      vii. Otherwise if *op* is /, then use the narrowest signed integer type whose width is larger than the width of T_1.

      viii. Otherwise use the integer type T given by the following rules:

         A. If either T_1 or T_2 has width 64, then the width of T is 64. Otherwise, the width of T is the narrowest width that is larger than the maximal width of T_1 and T_2.

         B. If either T_1 or T_2 is signed, then T is signed. Otherwise, T is unsigned.

Examples

```
constant a = { x = 0 } + 1 # Type is { x : U16 }
constant b = (1.0:F32) + 2.0:F32 # Type is F32
constant c = (1.0:F32) + 2.0 # Type is F64
constant d = 1 + 2.0 # Type is F64
constant e = 65535 / 2 # Type is U16
constant f = (100:U64) / -1 # Type is I64
constant g = 0xFFFFFFFF / -1 # Type is I64
constant h = 1:U64 + 2:U64 # Type is U64
constant i = 1:U64 + 2:I64 # Type is I64
constant j = 255 + 1 # Type is U16
constant k = 255 + 1:I8 # Type is I16
constant l = 1 + (-1) # Type is I32
```

In the last line, note that `-1` has type `I16` by the typing rules for integer literals and unary minus expressions.

Logical Expressions

To type a logical expression e_1 *op* e_2, the semantic analyzer does the following:

1. Check that e_1 and e_2 both have type `boolean`.
2. Assign the type `boolean` to the expression.

Membership Expressions

To type a membership expression e_1 `in` e_2, the semantic analyzer does the following:

1. Check that e_1 has type T_1 and e_2 has type T_2.
2. Check that T_2 may be converted to `set` T_1.
3. Assign the type `boolean` to the expression.

Equality Expressions

To type an equality expression e_1 `=` e_2, the semantic analyzer does the following:

1. Check that e_1 and e_2 have a common type *T*.
2. Assign the type `boolean` to the expression.

Approximation Expressions

To type an approximation expression e_1 `+-` e_2, the semantic analyzer does the following:

1. Check that e_1 and e_2 have a common type *T*.
2. Check that *T* is a numeric type.
3. Assign the type `range` *T* to the expression.

Structure Expressions

To type a structure expression `{ f_1 = e_1 , ... , f_n = e_n }`, the semantic analyzer does the following:

1. For each i \in [1, n]
    a. Compute the type T_i of e_i.
    b. Check that T_i is a structure member type. If not, throw an error.
2. Use the type `{ f_1 : T_1 , ... , f_n : T_n }`.

Examples

- The type of `{ x = 0, y = 1.0 }` is `{ x : U8, y : F64 }`
- The type of `{ a = [ 0, 1 ], b = { x = 0, y = 1.0 } }` is `{ a : [2] U8, b : { x : U8, y : F64 } }`
- The expression `{ x = 0..1 }` is not well-typed, because `range U8` is not a structure member type.

Array Expressions

To type an array expression `[ e_1, ... , e_n ]`, the semantic analyzer does the following:

1. Check that n > 0. If not, throw an error.
2. Compute the type T_i of each e_i.
3. Compute the common type T of the list of types T_1, ..., T_n.
4. Check that T is an array element type. If not, throw an error.
5. Use `[ n ]` T.

Examples

- The type of `[ 0, 1 ]` is `[2] U8`.
- The type of `[ 0, 65535 ]` is `[2] U16`.
- The type of `[ 0, 1.0 ]` is `[2] F64`.
- The type of `[ { x = 0, y = 0 }, { x = 1.0, y = 1,0 } ]` is `[2] { x : F64, y : F64 }`.
- The type of `[ [ 0, 1, 2 ], [ 3.0, 4, 5 ] ]` is `[2][3] F64`
- The expression `[ { x = 1 }, { y = 2 } ]` is not well typed, because `{ x : U8 }` is not compatible with `{ y : U8 }`.
- The expression `[ 0..1 ]` is not well typed, because `range U8` is not an array element type.

Range Expressions

To type a range expression e_1 `..` *e_2*, the semantic analyzer does the following:

1. Compute the type T_1 of e_1 and the type T_2 of e_2.

2. Compute the common type T of T_1 and T_2.
3. Check that T is a range element type. If not, throw an error.
4. Use the type `range` T.

Examples

- The type of `0..1` is `range U8`.
- The type of `0..65535` is `range U16`.
- The type of `0..1.0` is `range F64`.
- The type of `{x = 0, y = 0 }..{ x = 1.0, y = 1.0 }` is `range { x : F64, y : F64 }`.

Set Expressions

To type a set expression `set {` e_1 `,` \ldots `,` e_n `}`, the semantic analyzer does the following:

1. Check that n > 0. If not, throw an error.
2. Compute the type T_i of each e_i.
3. Compute the common type T of the list T_1, ..., T_n.
4. Check that T is a set element type. If not, throw an error.
5. Use `set` T.

Examples

Examples of sets containing integer types:

```
constant a = set { 0, 1, 2 } # Type is set U8
constant b = set { 0..1, 2 } # Type is set U8
constant c = set { (0..1)..(0..1) } # Error: this expression is not well typed
constant d = set { set { 0 , 1 } } # Error: this expression is not well typed
constant e = set { [ 0, 0 ]..[ 1, 1 ] } # Type is set [2] U8
constant f = set { [ 0, 0 ]..[ 1, 1 ], [ 2 , 2 ] } # Type is set [2] U8
constant g = set { { x=0, y=0 }..{ x=1, y=1 } } # Type is set { x : U8, y : U8 }
constant h = set { { x=0, y=0 }..{ x=1, y=1 }, { x=2, y=2 } } # Type is set { x : U8,
y : U8 }
```

Examples of sets containing floating-point types:

```
constant a = set { 0.0, 1.0, 2.0 } # Type is set F64
constant b = set { 0.0..1.0, 2.0 } # Type is set F64
constant c = set { (0.0..1.0)..(0.0..1.0) } # Error: this expression is not well typed
constant d = set { set { 0.0, 1.0 } } # Error: this expression is not well typed
constant e = set { [ 0.0, 0.0 ]..[ 1.0, 1.0 ] } # Type is set [2]F64
constant f = set { [ 0.0, 0.0 ]..[ 1.0, 1.0 ], [ 2.0, 2.0 ] } # Type is set [2]F64
constant g = set { { x=0.0, y=0.0 }..{ x=1.0, y=1.0 } } # Type is set { x : F64, y :
F64 }
constant h = set { { x=0.0, y=0.0 }..{ x=1.0, y=1.0 }, { x=2.0, y=2.0 } } # Type is
set { x : F64, y : F64 }
```

Identifier Expressions

To type an identifier expression e, the semantic analyzer resolves the identifier to a definition and uses the type given in the definition.

Example

```
constant a = 42 # a is a constant with type U8
constant b = a # The expression a refers to the constant a and has type U8
```

Dot Expressions

To type a dot expression e `.x`, the semantic analyzer does the following:

1. If e`.x` is a qualified identifier that represents the use of a definition according to the rules for resolving qualified identifiers, and the use is a valid dot expression, then use the type given in the definition.
2. Otherwise compute the type T of e and do the following:
   a. If T is a structure type with a member `x`, then use the type of the member.
   b. Otherwise if T is a range type `range` T', and T' is a structure type with a member `x`, and `x` has type T'', then use the type `range` T''.
   c. Otherwise if T is a set type `set` T', and T' is a structure type with a member `x`, and `x` has type T'', then use the type `set` T''.
   d. Otherwise e`.x` is invalid. Throw an error.

Examples

Example 1

```
module M {
  constant a = 0 # The constant M.a has type U8
}
constant b = M.a # Expression M.a represents a use of the definition M.a.
                 # Its type is U8.
```

Example 2

```
constant a = { x = 0, y = 1 } # The type of s is { x : U8, y : U8 }
constant b = a.x # The type of a.x is U8
```

Example 3

```
constant a = { x = 0, y = 0 }..{ x = 1, y = 1 } # The type of a is range { x : U8, y :
U8 }
constant b = a.x # The type of b is range U8
```

Example 4

```
constant a = set { { x = 0, y = 0 }..{ x = 1, y = 1 } } # The type of a is set { x :
U8, y : U8 }
constant b = a.x # The type of b is set U8
```

Array Index Expressions

To type a array index expression e_1 [*e_2*], the semantic analyzer does the following:

1. Compute the type T_1 of e_1 and the type T_2 of *e_2*.
2. Check that T_2 is may be converted to a primitive type. If not, throw an error.
3. If T_1 is an array type [n] T, then use the type T.
4. Otherwise if T is a range type `range` T', and T' is an array type [n] T'', then use the type `range` T''.
5. Otherwise if T is a set type `set` T', and T' is an array type [n] T'', then use the type `set` T''.
6. Otherwise e_1[e_2] is invalid. Throw an error.

Examples

Example 1

```
constant a = [ 0, 1 ] # The type of a is [2] U8
constant b = a[0] # The type of b is U8
```

Example 2

```
constant a = [ 0, 0 ]..[ 1, 1 ] : range [2] U8 # The type of a is range [2] U8
constant b = a[0] # The type of b is range U8
```

Example 3

```
constant a = set { [ 0, 0 ], [ 1, 1 ] }# The type of a is set [2] U8
constant b = a[0] # The type of b is set U8
```

Example 4

```
constant a = [ 0 , 1 ]
constant b = a[ { x = 0 } ] # Error: Cannot index an array with a structure
```

Parenthesis Expressions

To type a parenthesis expression (e), the semantic analyzer computes the type T of e.

Example

```
constant a = (1 + 2) * 3 # The type of the sub-expression 1 + 2 is U8
```

Explicitly-Typed Expressions

To type an explicitly typed expression e : t, the semantic analyzer does the following:

1. Compute the type T of e.
2. Check that T can be converted to t. If not, throw an error.

Examples

```
constant a = 1 : F32 # The type of the expression on the right-hand side is F32
constant b = 0 : { x : F32, y : F32 } # The type of the RHS expression on the right-
hand side is { x : F32, y : F32 }
constant b = { x = 0, y = 0 } : F32 # Error: Cannot convert a structure to F32
```

Type Conversion

We say that a type T_1 **may be converted to** another type T_2 if every value represented by type T_1 can be converted into a value of type T_2.

Here are the rules for type conversion:

1. Any primitive type may be converted to any other primitive type.
2. A named type may be converted to the representation type specified in the type definition to which it refers.
3. An enum type may be converted to a primitive type or to another enum type that refers to the same enum definition.
4. A structure type T_1 may be converted to another structure type T_2 if the two types have the same member names, and each member type of T_1 may be converted to the corresponding member type of T_2.
5. An array type T_1 may be converted to another array type T_2 if the two types have the same size (that is, the size expressions in the two types resolve to the same compile-time constant), and the element type of T_1 may be converted to the element type of T_2.
6. If T_2 is a range type and T_1 may be converted to the member type of T_2, then T_1 may be converted to T_2.
7. A range type T_1 may be converted to another range type T_2 if the member type of T_1 may be converted to the member type of T_2.
8. A range type T_1 may be converted to a set type T_2 if the member type of T_1 may be converted to the member type of T_2.
9. A set type T_1 may be converted to another set type T_2 if the member type of T_1 may be converted to the member type of T_2.
10. A primitive type T may be converted to a structure type T' if T may be converted to each member type of T'.
11. A primitive type T may be converted to an array type T' if T may be converted to the element type of T.
12. Type convertibility is transitive: if T_1 may be converted to T_2 and T_2 may be converted to T_3, then T_1 may be converted to T_3.

We say that two types T_1 and T_2 are **convertible to a common type** T if T_1 and T_2 both may be converted to T. We say that a list of types T_1, ..., T_2 is convertible to a common type T if T_1 and T_2 are convertible to a common type T'_2, T'_2 and T_3 are convertible to a common type T'_3, and so forth.

Computing a Common Type

Pairs of Types

Here are the rules for resolving two types T_1 and T_2 (e.g., the types of two subexpressions) to a common type T (e.g., the type of the whole expression):

1. If T_1 and T_2 are the same type, then let T be T_1.
2. Otherwise if T_1 and T_2 are both primitive types, then do the following:
    a. If T_1 and T_2 are both floating-point types, then use the floating-point type with the same

width as the maximal width of T_1 and T_2.

b. Otherwise if one of T_1 and T_2 is an integer type and the other is a floating-point type, then use the floating-point type.

c. Otherwise use the integer type T given by the following rules:

   i. If either T_1 or T_2 is signed, then T is signed. Otherwise, T is unsigned.

   ii. If either T_1 or T_2 has width 64, then the width of T is 64. Otherwise, the width of T is the narrowest width that can represent all the values of both types. For example, if one is `U16` and the other is `I8`, then use `I32`.

3. Otherwise if T_1 or T_2 is a named type, then replace each named type with its representation type and reapply these rules.
4. Otherwise if T_1 and T_2 are enum types that refer to the same definition, then use T_1.
5. Otherwise if T_1 is an enum type T, then replace T_1 with the representation type specified in the definition of T and reapply these rules.
6. Otherwise if T_2 is an enum type T, then replace T_2 with the representation type specified in the definition of T and reapply these rules.
7. Otherwise if T_1 and T_2 are structure types that are convertible to a to a common type, then T is the structure type that results from applying these rules to each pair of corresponding members of T_1 and T_2.
8. Otherwise if T_1 and T_2 are array types with the same size n and member types T'_1 and T'_2, then apply these rules to resolve T'_1 and T'_2 to T' and let T be `[`n`]` T'.
9. Otherwise if T_1 and T_2 are range types with member types T'_1 and T'_2, then apply these rules to resolve T'_1 and T'_2 to T' and let T be `range` T'.
10. Otherwise if T_1 and T_2 are set types with member types T'_1 and T'_2, then apply these rules to resolve T'_1 and T'_2 to T' and let T be `set` T'.
11. Otherwise if one of T_1 and T_2 is a set type and the other is not, then apply these rules to convert the non-set type to a set type, and then reapply these rules.
12. Otherwise if one of T_1 and T_2 is a range type and the other is not, then apply these rules to convert the non-range type to range type, and then reapply these rules.
13. Otherwise if one of T_1 and T_2 is a primitive type and the other one is a structure type S, then apply these rules to resolve the primitive type and each of the structure member types to a common type. Let T be the structure type whose members are the members of S and whose member types are the corresponding common types.
14. Otherwise if one of T_1 and T_2 is a primitive type and the other one is an array type `[` n `]` T', then apply these rules to resolve the primitive type and T' to a common type T''. Let T bet the array type `[` n `]` T''.
15. Otherwise the attempted resolution is invalid. Throw an error.

Lists of Types

To compute a common type for a list of types T_1, ... , T_n, do the following:

1. Check that n > 0. If not, then throw an error.

2. Compute the type T_1 of e_1.
3. For each i in [2,n]
    a. Compute the type T of e_i.
    b. Compute the common type T_i of T_(i-1) and T.
4. Use T_n as the common type of the list.

Evaluation

Evaluation is the process of transforming an expression into a value. In the current version of tnet, all evaluation happens during translation, either in resolving the size of an array type or in resolving a definition.

Values

A **value** is one of the following: an integer value, a floating-point value, a structure value, an array value, a range value, or a set value. Every value belongs to exactly one concrete type, and the types partition the values.

Integer Values

An **integer value** is an ordinary (mathematical) integer value together with a signed or unsigned primitive integer type. Formally, the set of integer values is the disjoint union over the primitive integer types of the values represented by each type:

- An unsigned integer type of width w represents integers in the range [0, 2^w-1]. For example, `U8` represents the integers [0, 255].
- A signed integer type of width w represents integers in the range [-2^(w-1), 2^(w-1)-1]. For example, `I8` represents the integers [-128, 127].

We write an integer value using the same notation as for tnet expressions. For example, we write the value 1 at type `U32` as `1 : U32`. The value `1 : U32` is distinct from the value `1 : U8`.

Floating-Point Values

A **floating-point value** is an IEEE floating-point value of 4- or 8-byte width. Formally, the set of floating-point values is the disjoint union over the types `F32` and `F64` of the values represented by each type:

- The type `F32` represents all IEEE 4-byte floating-point values.
- The type `F64` represents all IEEE 8-byte floating-point values.

We write a floating-point value using the same notation as for the corresponding tnet expression. For example, we write the value 1.0 at type `F32` as `1.0 : F32`.

String Values

A **string value** is a sequence of characters. We write string values using the same notation as for the corresponding tnet expression. For example, `"\"abc"` represents the character sequence `"abc`.

Structure Values

A **structure value** consists of a set of identifiers (the names of the structure members) and a mapping that associates a value to each identifier. We write structure values using the same notation as for structure expressions. For example, the structure value `{ x = 0 : U8, y = 1 : U32 }` has two members, `x` with value `0 : U8` and `y` with value `1 : U32`.

The type of a structure value is the structure type consisting of the member names and the types of the member values. For example, the type of `{ x = 0 : U8, y = 1 : U32 }` is `{ x : U8, y : U32 }`.

Array Values

An **array value** consists of a nonnegative integer size S and a list of values (the array elements) of size S. The array elements must all have the same type T.

We write array values using the same notation as for array expressions. For example, the array value `[ 0 : U8, 1 : U8 ]` has size 2 and elements `0 : U8` and `1 : U8`.

The type of the array value with size S and element type T is `[` S `]` T.

Range Values

A **range value** is a pair of values v_1 and v_2 where v_1 and v_2 are the respective values of the operands to the range expression e_1 `..` e_2.

We write range values using the same notation as for range expressions. For example, the range value `0:U8..1:U8` has size 2 and elements `0 : U8` and `1 : U8`.

The type of the range value with range member type T is `range` T.

Set Values

A **set value** consists of a list of values (the set elements). The set elements must all have the same type T.

We write set values using the same notation as for set expressions. For example, the set value `set { 0 : U8, 1 : U8 }` has the elements `0 : U8` and `1 : U8`.

The type of the set value with element type T is `set` T.

Concrete Types

The **concrete types** are the types of values. A concrete type is identical to a type appearing in a tnet source program, except as follows:

- Array types have nonnegative mathematical integer values instead of tnet expressions for their sizes. For example, `[ 2 ] U8` is a concrete array type.
- Enum types and named types are associated with their definitions, not their uses. There is only one concrete type per definition.

Evaluating Expressions

Evaluation of expressions occurs as stated in the expression descriptions. When evaluating a unary minus expression or binary arithmetic expression, the subexpression values are converted to a type wide enough to hold the result before carrying out the evaluation. For example, `1 : U8 + 255 : U8` evaluates to `256 : U16`.

Type Conversion

The following rules govern the conversion of a value v_1 of type T_1 to a value v_2 of type T_2.

Unsigned Integer Values

1. If T_1 and T_2 are both unsigned integer types and T_2 is narrower than T_1, then construct v_2 by truncating the unsigned binary representation of v_1 to the width of v_2. For example, `(0x1234 : U16) : U8` evaluates to `0x34 : U8`.
2. Otherwise if T_1 and T_2 are both unsigned integer types, then v_2 is the integer value of v_1 at the type of v_2. For example, `(0x12 : U8) : U16 = 0x12 : U16`.

Signed Integer Values

1. If T_1 and T_2 are both signed integer types and T_2 is narrower than T_1, then construct v_2 by truncating the two's complement binary representation of v_1 to the width of v_2. For example, `(-0x1234 : U16) : U8` evaluates to `-0x34 : U8`.
2. Otherwise if T_1 and T_2 are both signed integer types, then v_2 is the integer value of v_1 at the type of v_2. For example, `(-0x12 : U8) : U16 = -0x12 : U16`.

Integer Values of Mixed Sign

If T_1 and T_2 are integer types with one signed and one unsigned, then do the following:

1. Construct the value v by converting v_1 to the type T, where T is signed if T_1 is signed and unsigned if T_1 is unsigned, and T has the same width as T_2.
2. Construct v_2 by converting v to T_2.

For example, `(-1 : I8) : U16 = 0xFFFF : U16`

Floating-Point Values

We use the standard rules for IEEE floating-point values to convert among integer values to and from floating-point values and floating-point values to and from each other.

Structures and Arrays

We convert structures and arrays member by member. For example:

```
constant a = { x = 1 : U8, y = 2 : U8 } : { x : U16, y : U16 } # = { x = 1 : U16, y =
2 : U16 }
constant b = [ 1 : U8, 2 : U8 ] : [2] U16 # = [ 1 : U16, 2 : U16 ]
```

Converting Primitive Values to Structures and Arrays

We convert a value of primitive type to a structure or array value by converting the primitive value to each member of the structure or array and applying this rule recursively to any structures or arrays encountered. For example,

```
constant a = 0 : { x : [2] U32, y : F64, z : { a : U8, b : U16 } }
```

evaluates to

```
constant a = { x = [ 0 : U32, 0 : U32 ], y = 0 : F64, z = { a = 0 : U8, b = 0 : I16 }
}
```

Converting Values to Ranges and Sets

The following rules apply:

1. Any value v of range element type may be converted to the range value v `..` v.
2. Any range value v_1 `..` v_2 may be converted to the set value `set {` v_1 `..` v_2 `}`.
3. Any value v of set element type may be converted to the set value `set {` v `}`.

Converting to the Representation Type

If value v has named type T with representation type T', then v may be converted to T'.

Translation

The tnet language is intended to support a variety of translation strategies. For example, we need to generate C++ code and code for the MEXEC planner and scheduler. Additional code generation may be needed in the future.

Tools

The following tools for checking and translation currently exist:

- `tnet-check`: Check the validity of a tnet program, without performing any translation.

- `tnet-constants`: Extract the constant definitions in a tnet program into an ASCII table. The table is in a form that can be used by the task network translation in the [3X Autonomy flight software repository](https://github.jpl.nasa.gov/SystemsAutonomyRTD/isf-aut/tree/master/Projects/Autonomy/TaskNetworks).
- `tnet-enums`: Generate one ASCII table for each enum definition in a tnet program. Tools in the 3x Autonomy FSW repository can use the tables to generate C++ code.
- `tnet-arrays`: Generate one array type description for each array type definition in a tnet program. Tools in the 3x Autonomy FSW repository can use the tables to generate C++ code.
- `tnet-structs`: Generate one ASCII table for each structure type definition in a tnet program. Tools in the 3x Autonomy FSW repository can use the tables to generate C++ code.
- `tnet-states`: Generate one ASCII table for each state definition in a tnet program. Tools in the 3x Autonomy FSW repository can use the tables to generate C++ code.
- `tnet-timelines`: Extract all timeline definitions in a tnet program into an ASCII table. Tools in the 3x Autonomy FSW repository can use the table to generate XML input to MEXEC.
- `tnet-state-translator`: Generate one C++ state translator for each timeline definition.

Eventually we will replace these tools with tnet code generators.

For further information about these tools, see the README file in the tnet-lang repository.

Scalar Replacement of Structures and Arrays

MEXEC does not support structures and arrays. Therefore, before generating constant and timeline definitions, tnet applies a transformation that replaces all structures and arrays with corresponding sets of scalars. This transformation is called **scalar replacement**.

Constants

For structure and array constants, tnet uses the following strategy:

1. Replace each structure constant with a set of constants, one for each structure member.
2. Replace each array constant with a set of constants, one for each array index.
3. Apply the transformation recursively.

For example, after scalar replacement, this

```
constant a = [ { x = 0, y = 0 }, { x = 1, y = 1 } ]
```

becomes this (expansion):

```
constant a = [ { x = 0, y = 1 }, { x = 2, y = 3 } ]
constant a_0_x = a[0].x
constant a_0_y = a[0].y
constant a_1_x = a[1].x
constant a_1_y = a[1].y
```

and then this (replacement):

```
constant a_0_x = 0
constant a_0_y = 1
constant a_1_x = 2
constant a_1_y = 3
```

Timelines

For array and structure timelines, tnet uses the same strategy as for constants. In addition, tnet expands any range or set constraints on the timeline by selecting the appropriate members of the range or set of arrays or structures. For example, this:

```
type VectorF32 = { x : F32, y : F32 , z : F32 }
type State = { v1 : VectorF32, v2 : VectorF32 }
state s : State = 0
constant min : VectorF32 = -1
constant max : VectorF32 = 1
timeline s.v1 in min..max
```

becomes this:

```
type VectorF32 = { x : F32, y : F32 , z : F32 }
type State = { v1 : VectorF32, v2 : VectorF32 }
state s : State = 0
constant min : VectorF32 = -1
constant max : VectorF32 = 1
timeline s.v1.x in (min..max).x
timeline s.v1.y in (min..max).y
timeline s.v1.z in (min..max).z
```

By the semantics of the dot operator, the transformed code is equivalent to this:

```
type VectorF32 = { x : F32, y : F32 , z : F32 }
type State = { v1 : VectorF32, v2 : VectorF32 }
state s : State = 0
constant min : VectorF32 = -1
constant max : VectorF32 = 1
timeline s.v1.x in min.x..max.x
timeline s.v1.y in min.z..max.y
timeline s.v1.z in min.z..max.z
```