# Neural Network Exchange Format

The Khronos NNEF Working Group

Version 2.0.0 - Draft, Revision 6, 2025-05-14

# Table of Contents

# Chapter 1. Introduction

This chapter is Informative except for the section on Terminology.

This document, referred to as the NNEF Specification” or just the “Specification” hereafter, describes the NNEF: what it is, what it is intended to be used for, and what is required to produce or consume it. We assume that the reader has at least a rudimentary understanding of neural networks and deep learning. This means familiarity with the essentials of neural network terminology and operations.

## 1.1. What is NNEF

NNEF is a file format for describing information about neural network models. It encapsulates two key aspects of neural networks: network structure and network data. To describe network structure in a flexible way, NNEF introduces a domain specific language called *SkriptND*, which was developed to describe many aspects of neural network operators and graphs in a self-contained way, along with a set of standardized operations to express common neural network architectures. The language is naturally human readable and editable, independent of implementation details but sufficient to compile executable code from it. The network data, which form parameters of the network, is stored in a simple binary format.

Exchanging information about neural networks in a standardized format has become inevitable with the spreading of deep learning, as neural networks found their way from academic research to real-world industrial applications. With the proliferation of open-source deep learning frameworks and hardware support emerging for the acceleration of neural networks, the field faces the problem of fragmentation. The goal of NNEF is to provide a standardized platform for connecting accelerated neural network execution engines and available deep learning tools. Neural networks trained in deep learning frameworks may be exported to NNEF by mapping their operations and data representation to the operations and data representation of NNEF. NNEF can also be used as the starting point for training, by building the training model from an NNEF description, in which case no model conversion is required after training (apart from the trained model parameters). Furthermore, neural network compilers and accelerator libraries can consume, compile and ultimately execute NNEF models.

NNEF also aims to provide a distilled collection of deep learning operations that are widespread in successful neural architectures. It is the result of studying open-source deep learning frameworks such as Caffe, Torch, Theano, TensorFlow, CNTK, Chainer, and abstracting out the computations and data structures common to them. It mainly focuses on operations that are possibly efficiently implementable on various target hardware, such as massively parallelizable operations.

Although the main focus of NNEF is to be a central chain in the pipeline from deep learning frameworks to neural network compilers and accelerator libraries, we envision that the format may be used by intermediate tools in the future, for transforming neural networks in ways that are independent both from the training and the execution process, such as model optimization and visualization.

**The application programmers' view**

For an application programmer, NNEF is a standardized way to store and transfer neural networks. Given a neural network in NNEF format, and a compiler or library that is able to consume it, the application programmer need not worry about how the network was trained or how the underlying hardware will execute it, as long as it has the capabilities to do so. The application programmer may query the compiler or library whether it is capable of consuming the given model. However, a standardized API to do so, is out of the scope of NNEF.

**What NNEF is not**

NNEF is not an API (Application Programming Interface). It does not define an execution model for neural networks, and hence it does not define a standardized test suit for checking correct execution of neural networks described in NNEF format (no conformance tests). Even though, it does clearly define the semantics of operations supposing infinite arithmetics, defining correct execution of actual implementations would require finite arithmetics and underlying representations to be taken into account, which is out of the scope of NNEF. However, it is capable of describing such information (such as quantization).

Tools that produce or consume NNEF may have various APIs. However, importantly for application programmers, an NNEF consumer that intends to execute a neural network will most probably have functionalities to import and compile a network described in NNEF, and feed that network with data afterwards. However, the exact nature of this API is out of the scope of NNEF. One such API is described by the OpenVX Khronos standard’s neural network extension, along with an execution model of neural networks.

## 1.2. What is new in NNEF 2.x

NNEF 1.x has a simple high level syntax for describing neural network models on the graph level. However, the syntax of NNEF 1.x is lacking in a couple of areas, that makes it inadequate for describing individual operators completely, hence leaving gaps in the description of models. NNEF 2.x aims to fill those gaps by extending the syntactic capabilities. However, in order to do so, the syntax required a complete overhaul to concisely accommodate the required extra information, and in the end a new, more involved language was born, termed *SkriptND* (pronounced *script-end*).

The main areas where NNEF 1.x syntax is lacking are the following (and are interconnected):

- definition of the shape inference computation of operators
- definition of the runtime computation of primitive operators
- dynamic control flow constructs

NNEF 2.x aims to fix these shortcomings of NNEF 1.x syntax. SkriptND is a domain specific language for describing N-dimensional (ND) array (also called *tensor*) computation used in neural networks in a self-contained way, including many details of operators, such as shape inference and the runtime computation, in a compact and flexible way to enable parameterized variations of models.

The fundamental unit in SkriptND could be seen as a sub-graph, which may eventually be a single operator on ND arrays or a complete graph as well. It may have attributes, inputs and outputs with well-defined shapes, and it may define its computation by calling other sub-graphs (compounding) or via primitive computation defined by mathematical formulae at the scalar level (lowering).

Because all these aspects can be described using the language syntax, (almost) all operators, including custom ones, can be fully defined using SkriptND, and tooling can take advantage of that. For example, the syntax allows neural network compiler stacks to compile models all the way to runtime code, (mostly) without relying on manually written lowering passes or compiler intrinsics.

Just like any other format that describes neural networks, SkriptND has to be accompanied by a means to store model parameters. For that end, NNEF 2.x continues with using separate binary streams, the same way as NNEF 1.x did.

**What NNEF 2.x is not**

SkriptND is not a general purpose programming language. It is rather a domain specific one, focusing only on tensor computation and describing tensor operators; it is specific to parallelizable operations on N-dimensional data arrays of a few data types. While it is to some extent possible to express generic programs in SkriptND, including scalar computation (0-dimensional arrays) or sequential computation (making use of branching and looping), that is not its fundamental goal, hence it is not tuned for such purpose.

## 1.3. Specification Terminology

The key words **must**, **required**, **should**, **recommend**, **may**, **optional**, **can** and **cannot** in this document have special meaning and are to be interpreted with respect to the validity of NNEF documents. Their interpretation is similar to what is described in RFC 2119, however, instead of dictating what an implementation is required or capable of doing, they describe what a valid document constitutes and how to handle it.

http://www.ietf.org/rfc/rfc2119.txt

**must**

When used alone, this word, or the term **required**, means that the definition is an absolute requirement for the validity of a document. Software that aims to produce valid documents needs to adhere to the requirement. Software that consumes documents are advised to check that the requirement is met and give and error message if not. However, such checking is not strictly necessary if the software knows that the document is valid (for example a previous stage has already checked its validity). In such a case, the software may assume that a valid document meets such a requirement.

When followed by **not** (“**must not**” ), the phrase means that the definition is an absolute prohibition of the specification, and the prohibited item would invalidate the document.

**should**

When used alone, this word means that there may exist valid reasons in particular circumstances to ignore a particular recommendation, but the full implications must be understood and carefully weighed before choosing a different course. When followed by **not** (“**should not**”), the phrase means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label. In cases where grammatically appropriate, the terms **recommend** or **recommendation** may be used instead of **should**.

**may**

This word, or the words **optional** and **optionally**, means that an item is truly optional; it may or may not appear in a valid document. An implementation that processes valid documents must be prepared for both the presence and the absence of it.

The additional terms **can** and **cannot** are to be interpreted as follows:

**can**

This word means that the particular circumstance or situation is a possibility in a valid document.

**cannot**

This word means that the particular circumstance or situation is not a possibility in a valid document.

There is an important distinction between **cannot** and **must not**, as used in this Specification. **Cannot** means something the format literally is unable to express, while **must not** means something that the document would become invalid if the case would occur.

# Chapter 2. SkriptND Syntax

This description of SkriptND syntax will start with some examples to get the big picture and general structure of a SkriptND document. Afterwards, it will dive deeper into the details, explaining each syntactic construct and the related semantics with small examples. Finally, a formal definition of the syntax will be provided.

## 2.1. A first glance

SkriptND is a domain specific language for describing tensor computations, from the definition of *operators* to the whole *graph* level, using a unified syntax. SkriptND is able to describe:

- the *typed* attributes, inputs and outputs of an operator or graph
- the *symbolic* shapes of inputs and outputs of an operator or graph
- the shape and attribute *constraints* that make an operator or graph instance *valid*
- the *math formula* to compute each output of an operator
- the *composition* of operators and graphs from smaller operations

Furthermore, SkriptND makes it possible to define packages of operators that can be imported for later use.

Here is an example of the definition of the classic network AlexNet, using a set of predefined common operators:

```
import nn
import layout

graph AlexNet {
    @attrib {
        batch: int = 1;
        classes: int = 1000;
    }
    @input {
        input: real[batch,3,224,224];
    }
    @output {
        output: real[batch,classes];
    }
    @variable {
        kernel1: real[64, 3, 11, 11];
        bias1: real[64];
        kernel2: real[192, 64, 5, 5];
        bias2: real[192];
        kernel3: real[384, 192, 3, 3];
        bias3: real[384];
        kernel4: real[384, 384, 3, 3];
        bias4: real[384];
        kernel5: real[256, 384, 3, 3];
        bias5: real[256];
        kernel6: real[4096, 256, 5, 5];
        bias6: real[4096];
        kernel7: real[4096, 4096];
        bias7: real[4096];
        kernel8: real[classes, 4096];
        bias8: real[classes];
    }
    @compose {
        conv1 = nn.conv{padding=0, stride=4}(input, kernel1, bias1);
        relu1 = nn.relu(conv1);
        pool1 = nn.max_pool{padding=0, size=3, stride=2}(relu1);
        conv2 = nn.conv{padding=2}(pool1, kernel2, bias2);
        relu2 = nn.relu(conv2);
        pool2 = nn.max_pool{padding=0, size=3, stride=2}(relu2);
        conv3 = nn.conv{padding=1}(pool2, kernel3, bias3);
        relu3 = nn.relu(conv3);
        conv4 = nn.conv{padding=1}(relu3, kernel4, bias4);
        relu4 = nn.relu(conv4);
        conv5 = nn.conv{padding=1}(relu4, kernel5, bias5);
        relu5 = nn.relu(conv5);
        pool3 = nn.max_pool{padding=0, size=3, stride=2}(relu5);
        conv6 = nn.conv{padding=0}(pool3, kernel6, bias6);
        relu6 = nn.relu(conv6);
        flat1 = layout.flatten{axis=1}(relu6);
        conv7 = nn.linear(relu6, kernel7, bias7);
        relu7 = nn.relu(conv7);
        conv8 = nn.linear(relu7, kernel8, bias8);
        output = nn.softmax(conv8);
    }
}
```

The script starts with `import` statements to fetch some pre-defined operators. Then the definition of the main graph follows. Besides the definition of inputs and outputs along with their corresponding shapes, the example shows the usage of (compile-time) attributes to parameterize the graph with *batch* size and number of *classes*. Then the main part of the definition introduces the weight and bias *variables* and the actual *composition* of the graph from invocations of pre-defined operators.

But SkriptND does not stop at this level, it allows the detailed definition of the operators as well. Here is a simple example for the definition of a matrix multiplication operator:

```
operator matmul {
    @input {
        A: real[m,k];
        B: real[k,n];
    }
    @output {
        C: real[m,n];
    }
    @lower {
        C[i,j] += A[i,l] * B[l,j],
            i < m, j < n, l < k;
    }
}
```

The operator has two inputs `A` and `B` of `real` tensors whose shapes are captured in variables `m`, `n` and `k` for later use. It has a single result `C` with the resulting shape derived from that of the inputs. Note the use of the common extent `k` to describe the agreement of shapes among inputs. The math formula describes the summation with a compact index notation (`C` is implicitly initialized to the null element of the `+=` operator, `0` in this case).

The following more complex example describes a convolution operator. It shows the usage of (compile-time) attributes of varying length (`..` modifier, called parameter *packs*) and inputs / outputs of varying rank and the `optional` type modifier. Furthermore, it includes assertions for the validity of attributes / input shapes along with formatted error messages.

```
operator conv {
    @attrib {
        stride: int..(d) = 1;
        dilation: int..(d) = 1;
        padding: optional int..(d);
    }
    @input {
        input: real[b,c,is..(d)];
        filter: real[n,c,fs..(d)];
        bias: optional real[n];
    }
    @using {
        fd = (fs - 1) * dilation + 1;
        paddings = 2 * padding ?? (is \ stride - 1) * stride + fd - is;
        os = (is + paddings - fd) / stride + 1;
    }
    @output {
        output: real[b,n,os..];
    }
    @assert {
        stride > 0: "'stride' must be positive; got {stride}";
        dilation > 0: "'dilation' must be positive; got {dilation}";
        is + paddings >= fd: "padded input-size must be greater than (dilated) filter-
size;
                              got input-size={is}, filter-size={fs},
dilation={dilation},
                              total-padding={paddings}";
    }
    @lower {
        output[bi,ni,i..] = bias[ni] ?? 0.0,
            bi < b, ni < n, i < os;
        output[bi,ni,i..] += input[bi,ci,|stride * i + dilation * j - paddings / 2|..]
                             * filter[ni,ci,j..],
            bi < b, ni < n, ci < c, i < os, j < fs;
    }
}
```

The `@using` block introduces intermediate expressions used later. The `??` operator coalesces an optional expression with a fallback value in case an optional parameter is not provided. The math formula also showcases the possibility to initialize a contraction (`+=`) with a value other than its null element. Furthermore, when indexing tensors, the index expression within the `| |` operator is checked against the valid range of the indexed tensor and indices out of the bounds are omitted from the contraction.

The following sections describe the syntax and semantics in more detail.

## 2.2. Operator definitions

The central notion in SkriptND is an *operator*. Operators are introduced by the `operator` keyword. A *graph* is like a top level operator introduced by the `graph` keyword, with some differences in their usage, but their definition syntax is mainly identical. An operator is described with a list of *blocks*, each block name starting with the `@` symbol, and enclosed in `{}`. Each block is made up of a list of expressions terminated by `;`. The following blocks can be defined:

```
operator Op {
    @dtype {
        # generic data-types
    }
    @attrib {
        # compile-time (scalar) attributes
    }
    @input {
        # run-time (tensor) arguments
    }
    @using {
        # compile-time expressions (re)used within the operator definition
    }
    @constant {
        # tensor constants
    }
    @variable {
        # tensor variables
    }
    @output {
        # run-time (tensor) results
    }
    @assert {
        # operator invocation validity checks
    }
    @lower {
        # math formula describing implementation kernel
    }
    @compose {
        # list of operator invocations that this operator is composed of
    }
    @update {
        # variable updates
    }
    @quantize {
        # quantization information for tensors
    }
}
```

The order of the blocks may be arbitrary, but there are dependencies among them which must be taken into account when processing them, and the above order shows a valid processing order:

- The block `@dtype` introduces type names that may be referenced by the consecutive blocks in the above order.

- The blocks `@attrib`, `@input`, `@using`, `@variable` and `@output` introduce identifiers that may be referenced in all the consecutive blocks in the above order, hence they must be processed in the above order.
- The block `@update` may depend on intermediate values declared in the `@compose` block.
- All other blocks are independent of each other and hence may be processed in any order.

The blocks make use of various *types* and *expressions*. The following sections describe these basic syntactic constructs which allow concise definition of operators that may have complex interactions of their attributes and parameter shapes. Afterwards, the syntax and semantics of each block type is described in detail.

## 2.3. Type system

The words *list* and *pack* are more-or-less synonymous, however, this document will use *list* to denote a specific kind of packed expression (items enumerated within `[]`), whereas we will use *pack* to denote the concept of a sequence of items.

The type system consists of a number of concrete and abstract primitive types with possible modifiers:

- The concrete primitive types are `int`, `real`, `bool`, `str`. The type `real` is understood as a mathematical real number without prescribing any notion of precision. Furthermore, `int` is understood as a mathematical (signed) integer number, without any notion of representation size.
- The abstract primitive types are `type`, `arith` and `num` and can be used to impose restrictions in generics. The type `type` encompasses all types, the type `arith` denotes arithmetic types (`int`, `real`, `bool`) and the type `num` denotes numeric types (`int` or `real`).
- Each type may be *packed* to denote a sequence of values of the same type, denoted by the symbol `..`. The length of a pack may also be denoted by an identifier. For example, `int..(k)` is a `k`-long pack of `int`s. Pack are always unidimensional, *multidimensional packs are not supported*.
- Each type may be made *optional*, when preceded by the keyword `optional`.
- Each type may be a *tensor* type, when a type name is followed by a list of extents (shape) within `[]`. For example, `int[m,n]` is an `m`-by-`n` tensor of `int`s. The extents may themselves be packed, denoting variable-rank shapes (however, the ranks are always compile-time expressions). For example `int[s..(r)]` is a tensor of rank `r` of shape `s`, where `s` itself is a pack of extents.
- The modifiers can be combined, for example `optional int[m,n]..(k)` is a possibly null, `k`-long pack of `m`-by-`n` tensors of `int`s. The optional modifier pertains to the whole type, and not its items in case of tensors or packs. In case of tensor packs, the extent across a given dimension may be different for each item. This is denoted by placing the `..` modifier *before* the extent specifier, as in `int[m,..n]..(k)`. In this case, `n` is a pack of extents of length `k`, with one item corresponding to each tensor in the pack (of length `k`). However, the *rank* of each item in a tensor pack is necessarily the same (as there is no way to denote the type differently).

Note, that there is no explicit keyword to denote a *null* value for an optional type. Null values may emerge when an optional argument is not provided in an invocation, or via propagation of null values through expressions.

Expressions have well defined types, including whether they are packed, optional or tensor expressions.

## 2.4. Expressions

Expressions can be built recursively starting from literals and identifiers:

- Integer literals consist of digits only, e.g. `123`.
- Real literals contain a decimal `.`, or are written in scientific notation, e.g. `1.23` or `1e-5`.
- The keywords `true` and `false` denote `bool` constant literals.
- The keyword `inf` denotes a `real` constant literal (infinity).
- The keyword <code>pi</code> denotes a <code>real</code> constant literal (the value of &#x03C0;).
- String literals that are delimited either by `"` (double quotes) or `'` (single quotes). Longer string literals can be split into multiple parts (and so multiple lines) by writing multiple strings in quotes right after each other (with only white spaces between them); they are treated as a single long string.
- Identifiers consist of alphanumeric (ascii) characters and the `_` character, and must start with an alphabetic character or `_`. The identifier consisting of a single `_` is reserved for a special purpose.

Literals are always non-packed, non-optional, non-tensor expressions. Identifiers may denote (any combination of) packed, optional or tensor expressions.

The default value of any concrete primitive type can be denoted by the type name followed by `()`, such as `int()`. This also applies and is especially useful in generics (e.g `T()`), where the type itself is unknown a-priori and may stand for any concrete type. The default value is `0` and `0.0` for `int` and `real` respectively, `false` for `bool` and `""` for `str`.

The following expressions can be built from literals and other expressions recursively.

**Lists**

Lists are enclosed in `[]` and the items are separated by `,`, for example: `[1, two, 3]`. Items in an array **may** be of any type but **must** be of the same type, and including an optional item into the list makes the whole list expression optional.

Packed values **may** be used to form other lists, but they **must** be *expanded* into a list using the `..` notation, as in `[a, b.., c]`, where `a` and `c` are non-packed values, and `b` is a packed value. List concatenation can be achieved by expanding more than one packed expressions into a list, for example `[b.., d..]`, where `d` is also a packed value. As a consequence, packs and list expressions **cannot** be nested, expansion always flattens out packs to a single dimension.

Furthermore, it is possible to use expansion syntax after a non-packed item, in which case the item is duplicated multiple times, indicated by the length specifier after the `..` in brackets, which is **required** in this case, as in `[a..(5)]`, which is equivalent to writing `[a,a,a,a,a]`. The length specifier, **must** either be an `int` expression or optionally **may** be a `bool` expression. In the latter case, it behaves as if the expression was cast to `int`, i.e. if the expression is `true`, the item is included (once) in the list, while if the expression is `false`, the item is not included in the list (conditional expansion). Note, that in case of integer literal items, a space **must** be inserted between the literal and the `..` to avoid mistaking the literal as type `real`, for example `[1 ..(5)]`.

### Ranges

A list item may also denote a range of consecutive values, written as `[b:e:s]` which equals a list of values starting from begin `b` (inclusive) all the way to end `e` (exclusive), with a stride of `s`. The stride **may** be omitted, as in `[b:e]`, in which case it is understood to be `1`. The stride **may** also be negative. Ranges **may** be used together with other list items, as in `[0, 2:n, b..]`.

### Zip expressions

A list item may also denote a zip expression that takes its value by interleaving multiple packs of equal length. Let `a` and `b` be two packs of length `k`. Then `c = [(a,b)..]` is a pack of length `2 * k`, where the first item is taken as the first item of `a`, the second is taken as the first item of `b`, the third item is taken as the second item of `a`, the fourth item is taken as the second item of `b`, and so forth. In general, the zip expression is written as a tuple of a number of packs enclosed in `()` within a list expression, and **must** always be immediately expanded using the `..` notation. Unzipping a pack can be achieved either by indexing with stride (see below), as in `a = c[0::2]` and `b = c[1::2]` or by putting the tuple expression to the left-hand-side of the assignment (where the context allows), as in `[(a,b)..] = c`.

### Indexing

A packed expression can be indexed using `[]` and an index or range of indices, for example `a[i]`, where `a` **must** be a pack and `i` **must** be of type `int`. Negative indices are also **allowed**, which are understood as starting from the end, for example `-1` is the last item. The index expression **may** itself be a pack (of type `int..`), in which case the result is also a pack, taking all the items from the indexed pack, designated by the indices in the indexing pack. A special case of this is when the index is a range, denoted by `a[i:j]`. In this case, the start and the end indices **may** be omitted, and a step may also be provided after a second `:`. The step **may** also be negative, in which case the items are taken backwards. For example `a[::-1]` reverses the pack `a`. Let `r` be the rank of the indexed pack. When omitting start/end indices, if the step is positive, then start is understood to be `0` and the end is understood to be `r`, while if the step is negative, then start is understood to be the last item, at index `r-1` and end is understood to be the item before the first one, that would be at index `-1`. Note that explicitly writing index `-1` has a different meaning as explained above (the last item), thus indexing in reverse until the beginning of the pack can only be achieved by omitting the end index.

Furthermore, the index expression **may** also be of type `bool..`, in which case the indexed pack is masked with the index expression, taking the items where the mask evaluates to `true`. The length of the mask pack **must** be the same as the length of the indexed pack.

Expressions of type `str` **may** also be indexed. Indexing for strings results in a substring of either a single character or multiple characters from a possibly discontinuous range. Range indexing for strings results in a substring with characters in the given continuous range.

### Substitution

Substitution is an expression for replacing part(s) of a pack with another value and using the result. It builds on indexing syntax to denote which part to replace, extended by what to replace it with. In general, let's suppose that `a` is a pack, `i` is a pack of indices, and `b` is a pack of new values with the same length as `i`. Then the expression `a[i] <- b` is the pack that we get when replacing values in pack `a` at indices `i` with values `b`. The result length is the same as that of `a`. The type of `a` and `b` can be any type, but **must** be the same for both, while the type of `i` **must** be an `int...`.

As a special case, `i` and `b` can be a single (non-packed) index and value, respectively. In this case the type of `i` **must** be an `int` and `b` **must** not be packed either.

**Unary and binary operators**

Unary and binary operators (in the mathematical sense) may be applied to form math and logical expressions:

The following unary operators may be applied:

- `+` and `-` for arguments of type `int` and `real`
- `!` for arguments of type `bool` (negation)

The following binary operators may be applied (with their usual semantics if not noted otherwise):

- arithmetic operators `+`, `-`, `*`, `/` for arguments of type `int` and `real` (same type for both arguments)
- logical operators `&&`, `||`, `^` (xor) and `=>` (implication) for arguments of type `bool`
- comparison operators `<`, `<=`, `>`, `>=`, `==`, `!=` for arguments of any type (same type for both arguments)
- operators `<<` and `>>` mean minimum and maximum of arguments, respectively, for arguments of type `int` and `real` (same type for both arguments)
- operator `%` means modulo for arguments of type `int`
- operator `**` means exponentiation for arguments of type `int` and `real` (same type for both arguments)
- operator `\` means division with rounding upwards for arguments of type `int` (whereas the regular operator `/` rounds downwards)

Arguments of unary and binary operators may be packs, in which case the operations are taken element-wise, and the result is also a pack. When both arguments of a binary operator are packs, they **must** have the same length.

**Select operator**

The operator `? :` may be used to select between two values `a` and `b` depending on a condition `c`, as in `c ? a : b`. Furthermore, the operator may be used without a third argument (as in `c ? a`), in which case the result is an optional type with null value when the condition is not met. One or more arguments may also be packs, in which case the result is a pack. If the condition is a pack, the branching is taken element-wise, in which case all pack length must agree. When the conditions is not a pack, the other two arguments need not be of the same length.

**The identity operator**

The special keyword operator `is` can be used to test the identity of two expressions. It differs from the `==` operator in that operator `is` is always evaluated *in compile time*, while operator `==` may become a *runtime* comparison if the expressions are dynamic. Thus, the operator `is` compares two expressions symbolically (after simplification / transformation), while operator `==` compares their values. Let's consider two dynamic tensors whose shapes are only known to be `s1` and `s2`. In compile time, it is not possible to decide if they are the same, hence the expression `s1 is s2` is `false`. However, the expression `s1 == s2` may be true if in runtime they happen to have the same values.

In case of packed expressions `s1` or `s2`, `s1 is s2` is a packed expression, taken element-wise.

**Folding**

Fold expressions (also called reductions) take a pack argument and return a non-packed one by the repeated application of corresponding binary operators, and are denoted using the `..` symbol. For any (hypothetic) operator `@`, and pack expression `x`, the corresponding fold expression is `x @ ..`.

The following fold operators are available:

- arithmetic operators `+` and `*` for argument types `int..` and `real..`, designating sum (`x + ..`) and product (`x * ..`) respectively. The sum of the empty pack is defined to be `0`, and the product of an empty pack is defined to be `1`.
- logical operators `&&` and `||` for argument type `bool`, designating conjunction (`x && ..`) and disjunction (`x || ..`) respectively. The conjunction of an empty pack is defined to be `true`, while the disjunction of an empty pack is defined to be `false`.
- comparison operators `<`, `<=` and `>`, `>=` testing as argument of type `int..` or `real..` whether a pack is (strictly) ascending (`x < ..`, `x <= ..`) or (strictly) descending (`x > ..`, `x >= ..`) respectively, resulting in a `bool` value. An empty pack is defined to be sorted (both ascending and descending).
- comparison operator `!=` and `==` to test if items in a pack are unique (`x != ..`) or uniform (`x == ..`) respectively, resulting in a `bool` value. These comparison operators can be applied to any packed type. An empty pack is defined to be unique. An empty pack may or may not have a uniform value, see below.
- operator `:=` to acquire the uniform value of a pack, if exists (and can be derived in compile-time), or null otherwise (the result is of optional type). A uniform value may exist in various cases, with two root cases. First, an explicit list expression may enumerate the same value for each item, in which case the uniform value exists. However if the list is empty or has different items, the uniform value is null. Second, a repeated expression `[x..(n)]` is by construction uniform, hence the uniform value exists (`x`). The same is true for non-packed subexpressions of packed expressions that get broadcasted implicitly (for example in element-wise binary operators). In case of all other packed expressions, the optional uniform value can be derived from its components if exists by the semantics of the expression. For example, in case of a binary expression, if both the left and right hand side has a uniform value, then the whole expression has a uniform value.
- operators `<<` and `>>` to acquire the minimum (`x << ..`) and maximum (`x >> ..`) of a pack of type `int..` or `real..`. Since the minimum and maximum operators don't have a natural null element that can be represented universally across numeric data types, these operators result in `optional int` or `optional real` types, and return null value when an empty pack is folded.

Note, that arithmetic and logical fold operators are commutative and associative, so left and right folds are equivalent and hence not distinguished syntactically.

Comparison folds slightly abuse the concept of folding, in the sense that they are not the iterated application of the corresponding binary operator on the previous result and the next item (they cannot be, since their result type is not the same as their argument types), but rather pair-wise comparisons on consecutive items in the sequence (except for the operator `!=`, which performs all-to-all comparison). However, the concept of reducing a sequence into a single value is similar, hence the similar notation. Furthermore, the order of execution does not matter.

Furthermore, a fold expression naturally combines with the same binary operator to result in an expression that has an initial value to the fold, as in `x + .. + 42`.

**Cumulative Folding**

Some fold expressions have a *cumulative* counterpart, that map a pack onto another pack (of the same length) that contains all the intermediate steps of the folding. For any (hypothetic) operator `@`, and pack expression `x`, the corresponding cumulative fold expression is `x @ ...`, that is, *three dots* are used instead of the regular two.

The following fold operators are available:

- arithmetic operators `+` and `*` for argument types `int..` and `real..`, designating cumulative sum (`x + ...`) and cumulative product (`x * ...`) respectively.
- logical operators `&&` and `||` for argument type `bool`, designating cumulative conjunction (`x && ...`) and cumulative disjunction (`x || ...`) respectively.
- operators `<<` and `>>` to acquire the cumulative minimum (`x << ...`) and cumulative maximum (`x >> ...`) of a pack of type `int..` or `real..`.

Note that an empty pack maps onto an empty pack in all cases.

**Identity testing**

The keyword `is` can be used to test (in compile-time) whether two expressions refer to the same entity. For compile-time (attribute) expressions this is equivalent to testing equality of their values. For dynamic (shape) expressions, it is equivalent to testing whether the expressions are the same (not just their dynamic values). For expressions that result in a tensor object, it is equivalent to testing whether the expressions result in the same tensor object, which again is decidable in compile-time.

**Containment testing**

The keyword `in` can be used to test whether an item `x` is contained in a pack `a`, as `x in a`, resulting in a value of type `bool`. The item `x` may itself be a pack, in which case each item is tested for containment in the pack `a`, and the result is a pack of type `bool..`. The type of `x` and `a` **must** be the same, but can be any type.

**Null coalescing**

The operator `??` can be used to coalesce an optional value with a default value when the optional value is null. For example, for a value `a` of optional type, `a ?? b` results in `a` when `a` is not null, and `b` otherwise. Both arguments may be packs (may have different lengths since optionality is a property of the whole pack, not its items), in which case the result is also a pack. Both arguments **must** be the same type, `a` **must** be `optional` and `b` **must not** be `optional`.

### Null testing

The `?` operator can be used to test a value for being null, e.g. `?a` returns `true` if `a` is not null, and `false` if it is null. The argument may be a pack, however, the result is a non-packed `bool` value (since optionality is understood for the whole pack, not its items).

Null testing can be used in combination with the `?:` operator to promote an optional value to a non-optional one. Suppose that `x` refers to an optional value, then in the expression `?x ? x : 0`, the `x` on the true branch of the select operator is promoted to a non-optional value. Such promotion also works in case the condition is a conjunction, and one term in the conjunction is a null test, such as `?x && c ? x : 0`.

### Casting

Casting is supported only between arithmetic types `int`, `real` and `bool` using function-call syntax: `real(x)` or `int(x)` or `bool(x)`. String type cannot be cast to and from (but string formatting can be used to convert values of other types to strings). Casting the argument type to itself (no-op) is also allowed. Furthermore, declared generic type names can also be used for casting, such as `T(x)` (see generics below).

A special expression is casting `inf` or `-inf` to `int` type, that is `int(inf)` and `int(-inf)`. These expressions can be used to denote extremal values of operations working on integers, such as minimum/maximum searches. Compilers may map these expressions to appropriate integer values in the runtime system where such operations are performed (as opposed to the where the compilation is performed, which may have different integer representation from the runtime system).

### Built-in functions

Built-in functions may be used in expressions in various contexts. To differentiate them from operator invocations, they have a special syntax that marks their name between back-ticks, e.g. `name` .

The following built-in functions are supported with `real` arguments and `real` results: `sqrt` , `exp` , `log` , `sin` , `cos` , `tan` , `asin` , `acos` , `atan` , `sinh` , `cosh` , `tanh` , `asinh` , `acosh` , `atanh` , `round` , `floor` , `ceil` , `frac` . Furthermore, the following functions are supported with both `int` and `real` arguments and corresponding results: `abs` , `sign` .

### String formatting

A string literal may contain formatting information. Formatting is denoted by expressions within the special markers `{}`, whose value is substituted to the string. For example, if `a: int = 40`, then the expression `"a = {a + 2}"` results in the string `"a = 42"`. If the characters `{` and `}` are to be used within the string, they must be escaped with `\`.

**Implicitly defined identifiers**

Whenever a *non-packed* tensor identifier `x` is defined, two related identifiers `x.shape` and `x.rank` get implicitly defined alongside of type `int..` and `int` respectively. The value of `x.rank` will be the number of dimensions of `x`, while `x.shape` will contain the extents in each dimension. The length of `x.shape` will equal the value `x.rank`. Note, that such identifiers are not defined for packed tensor identifiers, since tensors in a pack do not necessarily have a uniform shape.

Whenever a *packed* tensor identifier `x` is defined, a related identifier `x.size` get implicitly defined alongside of type `int`. The value of `x.size` will equal the number of items in the pack.

**Tensor access**

Tensor access expressions are constrained to the math formulas describing tensor computation of an operator in the `@lowering` block.

Tensor access expressions are of the form `x[i,j,⋯]` where `x` is the name of a tensor, and `i`, `j`, etc. are index expressions (which themselves may contain any of the above expressions, including nested tensor accesses). The number of indices **must** equal the previously declared rank of the tensor `x`. Tensor index expressions **may** themselves be packed, in which case they cover multiple dimensions. The type of an access expression is the previously declared data type of the tensor `x`. If the tensor `x` is only 1 dimensional, then the access expression **must** contain an extra `,` after the index, as in `x[i,]`, to be able to differentiate it syntactically from regular pack indexing.

The name `x` **may** itself denote a pack of tensors. In this case, either an item of the pack **may** be selected before accessing it, as in `x[k][i,j,⋯]`, where `k` is the pack index, or the tensor access **may** pertain to all items in a pack (simply writing `x[i,j,⋯]`), in which case the result itself is a pack, whose each item is formed by accessing each pack item with the same indices.

Tensor accessing **may** be explicitly guarded against indexing out of bounds. This is achieved by placing the corresponding index expression within `| |`, as in `x[|i|,⋯]`. Furthermore, the index may be remapped in case it falls out of bounds on either end. This is denoted by the expression `|i <> i0 : i1|`, where `i0` is the index expression used if `i` becomes negative, and `i1` is the index expression used when `i` becomes larger or equal to the size of the tensor in the given dimension. See Defining lower level computation for further details on how such expressions are used.

### 2.4.1. Null propagation

Expressions that are not specifically designed for handling optional types (all expressions except null coalescing and null testing) *propagate* optionality and the null value. That is, the result type of such an expression that contains an optional type becomes optional itself, and if a null value is encountered during evaluation for any sub-expression, the result is null as well. For example, if any arguments of a unary, binary or select operator are null, the result is null. If any items in a list is null, the result is null. However, the result of null testing and null coalescing is never null (and the result type never optional).

As will be seen below, null propagation may have effect beyond the expressions themselves, depending on the context in which optional values are used. In a larger context, sub-expressions that evaluate to null are typically omitted, this way allowing their conditional processing. For example, assertions that become null are not checked, as it would be meaningless. This way, it becomes easy to write them, only focusing on the cases when they do have a non-null value. Further examples of null propagation will be indicated in the appropriate sections.

## 2.5. Operator attributes

Attributes of an operator are introduced by the `@attrib` block. Each item in the block declares a single attribute. A declaration **must** consist of a name, a type specifier (which **may** include a pack length), and **optionally** a default value. The type **must not** be a tensor type (it **must not** have a shape specifier). For example,

```
@attrib {
  a: int = 42;
}
```

introduces and attribute `a` of type `int` with default value `42`. Similarly,

```
@attrib {
  a: int..(k) = [];
}
```

introduces and attribute `a` of packed type `int..` whose length is denoted `k`, and the default value is an empty list. The length `k` may be omitted if it is not used throughout the operator definition, by writing `a: int..;` (in which case it will become an implicitly defined symbol named `|a|` used in error messages for example).

It is also allowed to use a non-constant expression as a default value that depends on another attribute or a well defined shape component, for example

```
@attrib {
  r: int;
  a: int.. = [0:r];
}
```

In this example, attribute `r` is well defined, so attribute `a` can use it in its default value, which is a range from `0` to `r` in this case. Such attributes are called *deferred*, because their evaluation may depend on other parameters (in case no attribute value is provided for `a` upon invocation and the default value needs to be used). Another such example would be a uniform default value, as in `a: int..(r) = [0 ..(r)]`, which means `0` repeated `r` times. This is a ubiquitous example, and when the pack length of `a` is marked and well defined, a simplified notation is allowed using a single non-packed value, as seen below:

```
@attrib {
  r: int;
  a: int..(r) = 0;
}
```

This notation has the advantage that the uniform attribute value expression does not explicitly depend on the variable r (the dependence is implicitly present however), and hence such an expression also works for attribute values explicitly assigned in the *invocation* of the operator (as opposed to writing a default value within the *definition* of the operator). Hence, in such cases, it is allowed to invoke an operator by assigning a single non-packed value in place of a packed attribute.

Attributes of optional types have an implicit default value of null, and they **cannot** have an explicit default value. For example

```
@attrib {
  a: optional int;
}
```

introduces and optional attribute a of type int, which will be null if not provided upon invocation of the operator.

Furthermore, it is also allowed to use an *optional* expression as deferred default value for a non-optional attribute. In that case, if the default value expression evaluates to a non-null value, then it can be used (if the attribute value is not provided). Otherwise, the attribute value **must** be explicitly provided.

```
@attrib {
  r: optional int;
  a: int..(k) = [0:r];
}
```

In this example, if attribute r is provided, the expression [0:r] is non-null, hence attribute a need not be provided (but can be, to override the default value). However, if r is not provided, the expression [0:r] will evaluate to null (due to null propagation), hence attribute a must be provided. This way, it is possible to create dependencies between attributes such that at least one of two attributes must be provided (or both).

## 2.6. Operator inputs and outputs

Inputs and outputs of an operator are introduced by the @input and @output blocks respectively. Each item in the block declares a single input / output. A declaration **must** consist of a name and a type specifier that **may** have a shape specifier in square brackets [] (but the shape specifier **may** be omitted in some cases). For example,

```
@input {
  x: real[m,n];
}
```

declares an input named x of type real, whose shape is m-by-n. Shape components **may** themselves be packed. For example,

```
@input {
  x: real[s..(d)];
}
```

declares a d-dimensional input x, where d can be arbitrary. If the length d is not used throughout the operator definition, it **can** be omitted, by writing `x: real[s..];`. In case the length specifier is a bool expression of known identifiers, such as attributes, the presence or absence of the item s is conditioned on the truth value of the expression d, and the deduced type of the item s is `optional int`.

When the input shape is captured into multiple components, it may be necessary to capture the rank of the entire shape as well. This can be done using the notation `^( )` after the type specifier, and before the shape specifier:

```
@input {
  x: real^(r)[s..(d),z..];
}
```

This way, both the rank of x is captured into r, and its components s and z are captured separately, with corresponding sub-ranks, if any (d here).

A shape component **may** as well be an optional expression (depending on an attribute for example). In case the expression evaluates to null, the component is omitted from the shape specification. This may be one way (besides packed shape components) to declare input/output shapes whose rank varies depending an attribute.

Using the same identifier for multiple shape components prescribes agreement of shapes among inputs, such as:

```
@input {
  x: real[m,k];
  y: real[k,n];
}
```

Inputs **may** be optional and **may** be packed. When packing tensors, some of their extents **may** be shared, while others **may** be distinct for each item in the pack. By default, using an extent identifier (either packed or non-packed) results in sharing the same shape components for all tensors in the pack. In order to denote a (single, non-packed) component that is not shared by all items in the input pack, the symbol `..` must be *prepended* to the extent. For example,

```
@input {
  xs: real[z, s..(l), ..t]..(k);
}
```

declares a pack of k inputs named xs that have shared extents z and `s..(l)` (of length l), but distinct extents t for each item in the input pack. As a consequence, t itself is a pack of length k, having a single distinct component corresponding to each item of the pack xs.

An extent and a pack length **may** be any expression, not just an identifier. If it is an *affine (linear) expression* of an identifier that has not been used before, it is treated as a declaration of that identifier (of type int or int..), otherwise the expression **must** consist of already declared identifiers and it is treated as a prescribed value for the given extent or pack length (that the input shape will be checked against).

An *affine expression* is an expression in the form a * x + b (or an equivalent one), where a and b **must** be constant expressions composed of constant literals and x is a variable symbol. The essence of an affine expression is that when a concrete value is bound to it, the value of the free variable x can be deduced easily and unambiguously. For example, if we know that 2 * x + 1 == 5, it is easy to deduce that x == 2. This can be used to define attribute pack lengths and shape extents and ranks:

```
@attrib {
  n: int;
}
@input {
  x: real[m, k + 1, s..(2 * d), n / 2];
}
```

In the above example, n is declared as an int attribute, and hence it can be used when prescribing the shape of x. The first three shape items (and its rank in case of the third item) are various affine expressions of previously unused variables m, k, s and d, whose values can be deduced from the actual shape of the tensor substituted to input x, and the last dimension is bound (checked against) by the expression n / 2. Note, that the expression 2 * d prescribes that the rank of s must be *even*.

Inputs **may** have default values, though as default values **must** be compile-time expressions, they are limited to 0-rank tensors. Packed inputs **may** have a list expression as default value, each item defining the value of an item (0-rank tensor).

In case of outputs, all identifiers in extent expressions **must** be declared beforehand, so that the output shapes can be evaluated given all the attributes and input shapes, as in the following example.

```
@input {
  x: real[m,n];
  y: real[k,n];
}
@output {
  z: real[m + k, n * 2];   # 'm', 'k' and 'n' has been declared in @input
}
```

Expressions in the output shape may be optional. If an expression evaluates to null (potentially due to null propagation), the corresponding dimensions are omitted from the output shape. This way it is possible to make output dimensions conditional (note that the ? : operator can also be used to explicitly generate an optional value depending on a condition by omitting the : and the third argument).

**Dynamic input and output shapes**

Operators **may** define outputs whose shape varies across certain dimensions, depending on the data in the input tensors. Furthermore, graphs **may** define inputs whose shape can vary. Even though such shapes will be dynamic, each shape **must** also define an *upper bound*. This static upper bound requirement keeps the otherwise undefined shape well-defined for *certain* purposes.

Input shapes of operators are naturally described by symbolic variables that capture the actual shape, as describe above. This same notation may be used in case of graphs as well, with the extra notation of the static upper bound after the `|` symbol. For example,

```
graph G {
  @input {
    x: real[s|10];
  }
  @output {
    y: real[s];
  }
}
```

denotes a dynamic input shape of a single dimension `s` with an upper bound of 10. The symbol `s` can then be used to propagate the shape to the output. Furthermore, the symbol `s` can also be used to prescribe agreement between two dynamic inputs, as below:

```
graph G {
  @input {
    x: real[s|10];
    y: real[s];
  }
}
```

In this case, the upper bound need not be specified again for the second input, but if present, it **must** be the same as in the first case. Such agreement constraints and upper bounds prescribe runtime checks for input shapes, as they can never be evaluated in compile-time.

Since graphs **must** always have a body composed of existing operators, their output shapes can always be derived. Hence, the output shapes can be omitted in case of graphs. This can be written component-wise, such as `output: real[~, ~, ~]`, in which case the number of dynamic components is explicitly marked (to be 3 here), or the entire shape definition can be omitted, leaving the rank also unmarked, such as `output: real`. In both cases it can be derived anyway.

Because of omitted output shapes, it may also happen that the input shape symbols are not used, hence, they can also be omitted (component-wise) for graphs. However, in case of the *main* graph, the upper bound must be defined. For example: `input: real[~|10, ~|20]` declares a dynamic input of rank 2, with upper bounds 10 and 20. The components may be packed, but the rank **must** be defined, such as `input: real[s|10 ..(3)]`, or even `input: real[~|10 ..(3)]`, both of which declare a dynamic input of rank 3 with upper bound 10 for each component. In case of subgraphs (not the main graph but invoked by a dynamic control flow construct), the input shape components can be omitted (~) without specifying the upper bound, and even the complete input shape may be omitted; it can be derived from the invocation anyway.

Omitting shape specifiers can be beneficial for tools that need to auto-generate sub-graphs and might not be able to explicitly specify input and output shapes.

The same syntax can be used for defining operators whose output shapes are dynamic because they depend on the actual data. As in that case there is no way to express the actual output shape, it is mostly left undefined (~), only the upper bound **must** be defined.

```
operator Op {
  @attrib {
    k: int;
  }
  @input {
    x: real[n,m];
  }
  @output {
    y: real[~|k,m];   # at most k outputs (in first dimension)
  }
}
```

However, an explicit symbol **may** be used to declare agreement of two or more dynamic output shapes, as s does in the following example:

```
@output {
  y: real[s|k,m];
  z: real[s|k,m];
}
```

Such information can be used to derive equivalence of dynamic shapes and expressions referencing such shapes.

Further usage information of dynamic shapes and bounds will be detailed in Dynamic tensor shapes.

### 2.6.1. Binding order of input shapes

If an input shape has more than one flexible-length parts (packed extents with previously unbound length), it would possibly be ambiguous to bind concrete values to extent identifiers upon invoking an operator. For example,

```
@input {
  x: real[s..,t..];
}
```

Here, when substituting a tensor of a given rank to x, it is not clear what proportion of the rank should be bound to s, and what to t. To remedy such ambiguities, attributes (of type int) and agreement with other shapes can be used to control the length of the first pack(s). For example, using an attribute:

```
@attrib {
  r: int;
}
@input {
  x: real[s..(r),t..];
}
```

In the above example, the first r number of dimensions are bound to s, and the rest to t. This implies, that the tensor substituted to x must have at least r number of dimensions.

The order in which inputs are bound may also matter in resolving ambiguities. Consider the following example, in which the first input can be of higher rank than the second:

```
@input {
  x: real[s..(d),t..];
  y: real[s..(d)];
}
```

When trying to bind extents to s and t based on input x, the binding is ambiguous, since d is not defined as an attribute. However, if first the input y is bound, it determines the value of rank d, which in turn makes the length of t unambiguous as well when binding input x.

Note, that ranks of packed attributes **cannot** be used to resolve ambiguities. This is because such attributes may be assigned a single non-packed value, in which case the length of the pack may not be known. Consider the following example:

```
@attrib {
  a: int..(d) = 0;
}
@input {
  x: real[s..(d),t..];
}
```

The above example is **invalid**, because the rank d **cannot** be determined for example if attribute a takes its default (uniform) value. Furthermore, optional attributes **cannot** be used for disambiguation either for similar reasons.

For the above reasons, a *binding order* is defined for inputs of each operator to eliminate such ambiguities when binding concrete values to symbolic shape components. The binding order only depends on the operator definition itself, not on any concrete invocation of it. This way, if a binding order is possible, it will work for *any* concrete invocation.

Given the values of *non-packed*, *non-optional*, *non-deferred* attributes (if any), the binding order of inputs is determined as follows:

- In the order of declaration, each *non-optional* input is examined if its shape components can be bound unambiguously. Binding is unambiguous if there is *at most 1* packed item in its shape, whose length is an affine expression of an identifier not yet bound by other attributes and inputs. If an input's shape can be bound, that input takes the next position in the binding order.
- Looping over the remaining unbound *non-optional* inputs in the order of declaration, the above check is repeated until new inputs are found that can be bound unambiguously and added to the binding order.
- Finally, the *optional* inputs are also checked, and if there are any remaining inputs, either *optional* or *non-optional*, whose shapes cannot be bound unambiguously, the operator definition is **invalid**. Otherwise, *optional* inputs are listed last in the binding order (in the relative order of their declaration).

After the binding order for inputs is established, packed attributes are also checked. If an attribute has a default value that is a non-constant expression (deferred attribute), it is checked if it can be evaluated given all the known symbols. If any of such checks fails, the operator definition is **invalid**.

Let the following operator definition serve as an example:

```
@attrib {
  r: int;
  a: optional int;
  b: int..(d) = 0;
  c: int = 2 * d + r;
}
@input {
  x: real[s..(r),z..(d),t..];
  y: real[s..(r),z..(d)];
}
```

First, the attributes are examined. The symbol `r` is registered as known; it can be used in shape component expressions hereafter. Symbol `a` is optional, and symbol `b` is packed, while `c` has a non-constant default value, hence those symbols are not yet registered as known.

Next, the first input `x` is examined. Since `r`, the length os `s` is known, it has 2 flexible-length shape components `z` and `t`, and as the value of `d` is not yet bound, it cannot be disambiguated. Then the next input `y` is considered, which only has 1 flexible-length component `z`. So input `y` will be the first in the binding order, which determines `d`, the length of `z`. In the next iteration, input `x` is examined again, this time however, it contains only 1 flexible-length shape component `t`, so it takes the second position in the binding order. No further unbound inputs remain.

Finally, attributes `a`, `b` and `c` are revisited, and it is found that the pack length `d` is now bound, hence `b` can be evaluated even if a uniform value is assigned to it. As `a` is optional but not packed, it is not considered further, and the default value of `c` is also checked, and found to be well defined as `d` and `r` are already defined.

**Evaluation and binding during an actual invocation**

If a binding order is found and the operator definition is valid, symbols in an *actual invocation* are bound in the following sequence:

1. Attribute values are evaluated and bound to the attribute names. Default values are only substituted if they are constant expressions (and the invocation does not assign a value to the attribute).
2. Going through the inputs in the (already known) binding order, each input component is bound to the actual shapes. The binding order ensures that each component (including pack lengths) can be either deduced (if it is an affine expression), or evaluated and checked against the substituted values (if it is not an affine expression). If the input is optional and its substituted value is null, the corresponding shape component symbols are assigned null values if not assigned any value previously. Note that such inputs are ordered last in the binding order.
3. Attributes whose default values are non-constant expressions are evaluated if they are not assigned a value by the invocation.
4. Lengths of packed attributes are checked if already known or deduced if not yet known. Packed attributes that take a non-packed value are converted to a uniform packed value if the pack length is known. If the pack length is not known, such an invocation is **invalid**.

If the deduction of any symbol in an affine expression is ambiguous (deduced to two different values), the invocation is **invalid**. Also, if non-affine pack length or shape expressions that are evaluated and checked against the actual input shapes do not match, the invocation is **invalid**.

At this point all symbols are well defined, and hence all expressions such as asserts and output shapes can be evaluated.

## 2.7. Variable and constant tensors

Variable tensors within a graph or an operator **can** be introduced in the `@variable` block, while constant tensors **can** be introduced in the `@constant` block. The syntax **must be** the same as that of inputs / outputs, with the constraint that their shapes **must** be expressions of already declared identifiers (as attributes or input extents) or (compile-time) constants.

```
@attrib {
  b: int;
}
@input {
  x: real[n];
}
@constant {
  c: real[b] = 0.0; # 'b' is already declared
}
@variable {
  w: real[b,n];   # both 'b' and 'n' are already declared
}
```

The expression after the `=` sign defines the value of the constant. Uniform constants can be non-packed expressions, while packed expressions (such as a list) **can** define a different value for each tensor element in a flat, row-major layout. Per-element initialization of constants whose shape is parameterized can also be performed using expressions that rely on *index symbols* that loop over the dimensions of the tensor. These index symbols are local to the declaration and are introduced with a `,` after the expression in the form of `id < bound` where identifier `id` is the index symbol, and expression `bound` is its upper bound (exclusive), while the lower bound is implicitly 0 (inclusive). For example, to declare an identity matrix constant, one can write:

```
@attrib {
  n: int;
}
@constant {
  I: real[n,n] = i == j ? 1.0 : 0.0, i < n, j < n;
}
```

It is also possible to define a pack of constants. The pack length **must** be explicitly defined, and in that case the value **must** be defined by a list of values, whose length equals the pack length, each item defining one constant in the pack. In this case, the (possibly different) value of each constant can only be uniform (though their shapes may vary).

Variables **must not** have their values explicitly defined, rather they are stored in separate data files mapped to the variables by their (possibly scoped) names. Updates to variables **can** be written in the `@update` block (see below).

## 2.8. Operator validity checking

The definition of an operator **can** also tell which range and combination of input shapes and attributes are valid. This can be defined in the `@assert` block, which is a list of boolean expressions (conditions) that **must** evaluate to `true` in order for the invocation of the operator to be **valid**. Optionally, an error message (formatted string) **may** be provided to be printed in case a condition is violated, followed with a comma separated list of expressions to be printed as further debug info:

```
@attrib {
  a: int;
}
# other blocks
@assert {
  a > 0: "a must be positive", a;
}
```

If the operator is called with an attribute `a` that is not positive, the above error message will be printed, along with the value of `a`. By default, the printed values are labelled with the exact expression written in the source; the above case would result in something like `a = <value of a>` being printed. However, that is not always desirable, as the printed expression may not be meaningful for debugging purposes (for example if it uses identifiers internal to the operator definition). For these cases, the label can be overridden with something more interpretable by writing `label: expression` instead of just the expression to be printed.

An assert condition **may** also be packed, in which case all of its items **must** evaluate to `true` (as if an `&& ..` folding was applied to the expression). If the value of an expression is null (because of null propagation taking place if one symbol in the expression is null), then the condition is skipped. This way it is possible to check an optional attribute only if it is actually provided.

## 2.9. Using helper symbols for intermediate expressions

In some cases it may be cleaner (or even required) to give a sub-expression its own identifier. This **can** be done in the `@using` block, which is a list of `id = expression` assignments. In these expressions, identifiers **must** be previously declared attributes and input extents:

```
@attrib {
  a: int;
}
@input {
  x: real[s..(d)];
}
@using {
  c = a + d;
}
# c can be used in other blocks such as @output
```

The expressions in the using block of an invoked operator are evaluated (in the order of declaration) right after all attributes and input shapes (including ranks) are evaluated. Symbols declared in the `@using` block are not visible in the `@attrib` and `@input` blocks.

In some cases, when the expression is a pack, its length may not be possible to evaluate symbolically, purely based on the declaration of the operator, and not its concrete instantiation. In order to allow some symbolic checks to run ahead of graph composition time, pack lengths **can** be hinted after `..` notation, within brackets. Hints are also allowed even when the rank of the right-hand-side can be evaluated symbolically, in this case the two **must** to match.

```
@using {
    c..(n) = ...;
}
```

As a special case, the left-hand-side of the `@using` declaration **can** be a list expression, in which case the list is decomposed to its elements or sub-lists. Furthermore, zip expressions **can** also be used in this context to *unzip* a list.

```
@using {
  [a,b..] = c;
  [(e,f)..] = c;
}
```

The first example splits the head element `a` of the list `c` from the tail `b`, while the second example unzips the list `c` into two lists `e` and `f`. Optionally, the length of pack subelements **can** be provided after the `..` operator, in which case the length **must** to match the total length of the right-hand-side minus the length of the rest on the left-hand-side.

The expressions in the `` `@using `` block **must** be evaluated in the order of their declaration, since a subsequent expression **may** depend on a previously defined symbol.

**Interplay between intermediate expressions and validity checks**

There is an intricate dependency between helper symbols and validity checks: the symbols used in expressions of helper symbols may need to be checked first for the expression to be meaningful, but at the same time, helper symbols themselves **may** be used in validity checks. This makes it necessary to interleave the evaluation of validity checks and helper symbols definitions in compiler implementations. Let's look at the following example.

```
@attrib {
  a: int;
  b: int;
  c: int;
}
@using {
  r = a / b;
}
@assert {
  b > 0: "b must be positive", b;
  a % b == 0: "a must be divisible by b", a, b;
  r + c == 42: "some condition must be true for a/b and c";
}
```

In the above, the compiler cannot start with evaluating `r`, because that might lead to an error (such as a division by zero), if the wrong value is supplied for `b`, *before* it could be checked. Also, the compiler cannot start with evaluating all the assertions, since `r` needs to be calculated before the third check. So the evaluation **must** be interleaved. In this case, the first two checks needs to be performed first, then the definition of `r` needs to be evaluated (which is safe by this time), then the third check.

A compiler implementation can perform the following safe strategy for evaluation. Going through the list of helper symbols to be calculated in the order of their declaration, *before* evaluating the next helper symbol, the compiler should try to check *all* assertions (not yet checked) *that are possible to evaluate* given the already known symbols up to that point (not including the next helper symbol itself). This makes sure that all assertions that only pertain to symbols used in the definition of the next helper symbol will have been checked. After that, the helper symbol is safe to be calculated (assuming that the operator definition is sound). Acquiring the value of the next helper symbol may in turn make further assertions possible to evaluate. In the end, after all helper symbol definitions are evaluated, the remaning not yet checked assertions **must** also be evaluated.

To decide whether an assertion is possible evaluate, the compiler only needs to inspect the assertion expression to only contain already known identifiers (attributes, input shape symbols and already evaluated helper symbols).

Going back to the above example, before the definition of the first helper symbol `r`, the compiler should check the assertions. The first two can be evaluated, because they are only formed of attributes `a` and `b`, hence they are checked. The third one cannot be evaluated because it also contains they not yet know symbol `r`. After this, the symbol `r` is evaluated, which makes it possible to evaluate the third assertion in the end.

Note, that the order in which the assertions are written does not matter. The compiler should repeatedly try to evaluate all of them, before evaluating the next helper symbol, and after evaluating all helper symbols (if any).

## 2.10. Composing operators

Operators **can** be composed from other operators, and the composition **can** be described in the `@compose` block, which is a list of assignments, where the left-hand-side is a tuple of identifiers, and the right-hand-side is an invocation of an already defined operator or a compile-time expression resulting in a tuple of tensors (or packs of tensors). For example, the following operator implements multiply-add for vectors of length `n`, given the multiplication (`mul`) and addition (`add`) operators:

```
operator mul_add {
  @input {
    x: real[n];
    a: real[n];
    b: real[n];
  }
  @output {
    y: real[n];
  }
  @compose {
    z = mul(x, a);
    y = add(z, b);
  }
}
```

The `@compose` block **may** introduce intermediate tensors (`z` above), but each intermediate tensor **must** have a unique name. Furthermore, the block **must** contain an assignment for all declared outputs, and the shapes and types of the outputs calculated through the composition **must** be consistent with the declared shapes and types.

As can be seen from the example above, the invoked operators can take tensor arguments within `()`. Scalar arguments (defined by any compile-time expression) may also be used, which are understood as 0 dimensional constant tensors. Furthermore, attributes **may** be specified within `{}`, as a list of key-value pairs, where the key **must** be a declared attribute of the operator, and the value **must** be a compile-time expression:

```
@compose {
  z = op{a=true, b=[1,2,3]}(x, y);
}
```

An operator **may** have one or more results, and input arguments and results **may** be packs of tensors:

```
@compose {
  a, [b, c] = op(x, [y, z]);
}
```

In case an invocation has packed results, the length of such results **must** be explicitly denoted, as it could not always be derived easily from the operator invocation on the right hand side. This can be done similarly as in case of parameter declarations, after the `..` operator:

```
@compose {
  a..(n), [b, c..(k)] = op(x, [y, z]);
}
```

If optional attributes or (last) inputs are omitted, they become null. Alternatively, the special symbol `~` can be used to omit an optional input or output from the middle of the sequence. Using `~` on the left-hand-side has the benefit of not generating unnecessary names, which may help avoiding name clashes.

```
@compose {
  a, ~, c = op(x, ~, z);
}
```

**Labelling operator invocations**

Optionally, the actual instance of the operator **may** be given a label upon invocation by prefixing it before a `:`, as in `label: op(⋯)`. This label can later be used to reference the instance of the operator and the declared tensors within.

```
@compose {
  y = op1: op{a: 42}(x);
}
```

If `op` declares a variable of name `w` for example, it can be referenced as `op1.w`. Labels of nested operator invocations form a nested scope separated by the `.` symbol, such as `op1.op2.w`.

In most cases, labels are also auto-generated to alleviate the need for explicitly labelling all operator invocations. There are two main categories for auto-generating labels:

- If a composite operator consists of only one component, whose right-hand-side contains a single operator invocation or can always be reduced to such (for example a branching with static conditions), and the operator does not define any local tensors (such as variables or constants) then the scope of the containing operator is propagated to the invocation without adding a new component to the scope.
- If an invocation has only one identifier on the left-hand-side (single result, which may be packed), and the right-hand-side contains a single operator invocation or can always be reduced to such, then the identifier of the single result is used as an auto-generated label.

These cases cover most uses of operator invocations, so explicit labels are rarely required. Both cases can be overridden by explicit labels.

The labels can be used to access *certain* nested tensors in *certain* contexts. By default, only declared variables and constants are accessible, inputs, outputs and intermediate tensors are not. This can be changed by defining an operator as public, using the `public` keyword before the keyword `operator`:

```
public operator op {
  # all input, output and intermediate tensor names are accessible
}
```

The scoped identifiers can only be used in file names referencing variables and in the `@quantization` block, for referencing tensors when attaching quantization info to them. Scoped identifiers **must not** be used in expressions (such expressions would violate locality, as it would provide access to tensors declared potentially far away from where they would be used).

It is **recommended** to only use the `public` keyword on custom defined operators whose usage is controlled by the same entity that defines it, for example when a graph is broken into subgraphs for code clarity, reuse or for purposes of control flow. This is because tensors inside an operator definition **can** be implementation details, and once extra information, such as quantization info, is attached to such tensors, the internals of the operator can only be changed by risking breaking the code that uses it. *All standard operators are private, as their internal implementation is subject to change*. Luckily, only a handful of them uses intermediate tensors in their definition, so it does not make a major difference anyway.

Furthermore, a sub-graph used in control flow can also be defined as `graph` (instead of as `operator`). The difference is that a graph defines its own name scope, the name of the graph being the root. However, when invoking a graph, this can be overridden by an *explicit* label, in which case the name scope of the invoked graph will be the (possibly nested) scope leading up to the label. Auto-generated labels do not apply in this case.

### 2.10.1. Control flow

Apart from tensor expressions on the right-hand side in the `@compose` block making use of operators `?:` and `??` for *compile-time* branching, SkriptND provides two means of *dynamic* control flow:

branching an looping. Both of them involves invoking sub-graphs of operations in a conditional manner.

Syntactically, sub-graphs can be written using either *invocations* of operators that can essentially describe sub-graphs, or by writing *blocks* of operations embedded in a bigger context (that is, the `compose` block of a graph or an operator). Invocations are written just like in the regular case without control flow, while embedded blocks of operations are written within curly braces `{}`, just like the `compose` block of an operator or graph itself. However, such embedded blocks **must** explicitly denote which tensors they yield as result, written after the `yield` keyword, which **must** be the last statement of the block. Blocks can use any previously declared tensor or attribute from their context. The yielded results **must not** be optional expressions. For example, the following block uses the tensor `x` from its outer context, and yields the tensors `y` and `z` as results.

```
{
  y = op1(x);
  z = op2(y);
  yield y, z;
}
```

Intermediate results in a block can also shadow existing tensor names in the outer context. Just like invocations, blocks can also be prefixed with a label to introduce an explicit name scope for the intermediate tensors within the block, otherwise an implicit name scope may apply if possible (the same way as in case of invocations).

A block can be opened on the right hand side of an assignment (inside an outer block):

```
{
  y = {
    ...
    yield x;
  }
}
```

Actually, the right-hand-side of a simple assignment

```
{
  y = x;
}
```

is equivalent to an empty block that simply yields `x` (that may be any tensor expression).

### Branching

Branching can be achieved by an `if-then-else` expression, with optional `elif-then` branches inserted before the `else` branch. It takes (at least) three invocations or blocks; a *condition*, a *consequent* on the `then` branch and an *alternative* on the `else` branch. The addition of `elif-then` branches adds more condition and consequent pairs. For example,

```
@compose {
  z = if cond(x,y) then op1(x,y) else op2(x,y);
}
```

or similarly written using blocks:

```
@compose {
  z = if {
    c = cond(x,y);
    yield c;
  } then {
    r = op1(x,y);
    yield r;
  } else {
    r = op2(x,y);
    yield r;
  }
}
```

Note however a difference between the two ways of writing. The latter case results in blocks of single operations, while in the previous case, the blocks are implicitly the `compose` blocks of the invoked operators themselves. This difference may be reflected in the resulting internal representation when compiling such statements.

As noted above, a block may be empty and only yield a tensor from the outer context, in which case the syntax can be simplified by omitting the curly braces and the `yield` keyword altogether. In the above example, the condition could be factored out as follows:

```
@compose {
  c = cond(x,y);
  z = if c then op1(x,y) else op2(x,y);
}
```

Similar syntax can be used for the consequent / alternative blocks when they simply yield an already declared tensor.

If the condition is a compile-time expression (not involving tensors, only compile-time variables), the branching becomes static (can be evaluated and eliminated in compile-time).

**Promotion of optional values**

A special case of a compile-time condition is when an optional value is tested for being null. In this case, the optional value is promoted to a non-optional one inside the branch where it is tested. Suppose that `x` refers to an optional value, then

```
@compose {
  z = if ?x then {
    # x can be used as non-optional value on this branch
  } else {
    # x is null on this branch
  };
}
```

Such promotion also works in case the condition is a conjunction, and one term in the conjunction is a null test, such as

```
if ?x && c then { ... }
```

**Looping**

Loops come in various flavors, which can all be mixed, but to illustrate them it is best to describe them in their pure forms:

- Loops can be based on *scanning* through a pack of tensors (`for` loop), or based on a *condition* (`while` loop).
- Loops can be *parallel*, working independently on each item, producing a separate result in each iteration, or can perform *folding / reduction*, when a single value is iteratively updated.

However, common to all loops is the repeated invocation of an operator or block of operations (the loop body), introduced after the `do` keyword, as in `do op1(x,y)` or `do { z = op1(x,y); yield z; }`, with additional details introduced by further clauses within the loop statement. For the sake of simplicity, the examples below will use operator invocation syntax for the loop body / condition, but they can equally be written using block syntax if the body is composed of multiple operators, just as in case of branching.

**Scanning loops**

Scanning loops feed the body with items of one or more tensor packs, scanning through the packs in parallel. Syntactically, it is introduced by the `for` keyword followed by a (comma separated) list of `item-identifier : pack` pairs to be iterated over (such packs are called *scan inputs*). The item identifiers declare *loop local* tensors to refer to the corresponding items of the scan inputs in each iteration, and are not visible outside of the loop statement. Their names **may** shadow tensors existing in the outer context. All scan inputs **must** have the same length.

Each invocation of the operator or block results in one or more outputs, and the outputs of each step of the iteration are then packed to form one or more results (called *scan outputs*).

The following example takes a pack of `k` tensors `xs` and adds another tensor `z` to each tensor in the pack, resulting in another pack of tensors `ys`:

```
operator add_to_all {
  @input {
    xs: real[n]..(k);
    z: real[n];
  }
  @output {
    ys: real[n]..(k);
  }
  @compose {
    ys = for x : xs do add(x, z);
  }
}
```

**Folding loops**

A folding loop computes a single value out of a pack of values. To achieve this, it updates a loop variable in each iteration. Such loop variables, also called *loop carried dependencies* must be declared and initialized, and are implicitly understood to be updated by each iteration. Syntactically, this is achieved by enumerating such variables after the `with` keyword, as a (comma separated) list of `variable-identifier = expression` pairs, denoting initial values assigned to newly declared loop variable identifiers. Optionally, the identifier **may** be followed by a `:` and a data-type and shape specifier (same syntax as for operator inputs/outputs), which can be useful when the initializer expression is a compile-time scalar but the declared tensor needs to have a non-zero dimensional shape (uniform value). The loop carried dependencies are local to the loop statement, they can only be used in the loop condition and body, they are not visible outside of the loop statement. Their names **may** shadow tensors existing in the outer context. The `with` clause **must** precede the `for` clause.

The following example sums all tensors in an input pack `xs`, resulting in a single tensor `y` as output. It declares a loop variable tensor `sum` with initial value 0.0 (of data-type and shape `real[n]` to match the inputs):

```
operator add_all {
  @input {
    xs: real[n]..(k);
  }
  @output {
    y: real[n];
  }
  @compose {
    y = with sum: real[n] = 0.0 for x : xs do add(sum, x);
  }
}
```

Updating the loop-carried variables works as follows. If a loop has N carried variables declared in the `with` clause, the *first N outputs* of the looped operator or the first N yielded results of the loop body block correspond to the updated values of the loop-carried variables. All other outputs of the body become scan outputs. The final values of the loop carried variables, along with the scan outputs if any, are assigned to identifiers on the left-hand-side. The above example has one loop carried variable `sum`, one scan input `xs` and no scan outputs. The final value of `sum` is stored in `y`.

By having both loop-local variables, and scan inputs/outputs, parallel and folding loops are handled uniformly and can be mixed naturally.

**Iteration count and iteration index**

So far an explicit iteration count and index has not been introduced, as they are not required for the basic construct of iterating over packs, due to the syntax of the `for` clause. However, in some cases an explicit iteration count may be required, or the looped body may need to receive the current index.

To specify the iteration count explicitly, the `do` keyword **may** have it appended using the usual `..()` repetition notation. The iteration count **may** either be a compile-time expression or a tensor (of type `int` (of 0 dimensions) in both cases). Note that in the default case of a `for` loop when the loop count is not explicitly specified, it equals the (common) length of the scan inputs, which is actually a compile-time expression. However, an explicit loop count may also be specified when a `for` clause is not present (also see condition based looping below). Furthermore, a dynamic loop count (of tensor type) allows a partial iteration over the scan inputs (whose lengths are fixed in compile-time), depending on the actual tensor value. In this case, the length of the scan inputs **must** be an upper bound for the dynamic value of the loop count.

To expose the current index as an `int` tensor of rank 0, and identifier can be specified before the iteration count in the `..` clause, separated by a `->` symbol. If only the iteration index is needed to be declared explicitly without an iteration count, the latter can be omitted after the `->` symbol. The index identifier is local to the loop and **may** shadow other identifiers in the outer context.

As an illustration, the following loop adds numbers (`int` tensors of rank 0) up to a limit `k`, which may itself be a tensor:

```
@compose {
  sum = with s = 0 do..(i -> k) add(s, i);
}
```

As further example on looping, the following is an operator for computing matrix powers by iterated matrix multiplication (hypothetical `matmul` operator), where the loop count (the power) is provided as a tensor, hence the loop is dynamic (and has no scan inputs or outputs):

```
operator matpow {
  @input {
    A: real[n,n];
    p: int[];
  }
  @output {
    B: real[n,n];
  }
  @constant {
    I: real[n,n] = i == j ? 1 : 0, i < n, j < n;
  }
  @compose {
    B = with X = I do..(p) matmul(X, A);
  }
}
```

### Condition based looping

Condition based looping makes use of two sub-graphs; a condition and a looped body. It comes in two forms; `while-do` and `do-while` for pre and post-testing respectively. In order to achieve meaningful computation, they typically need to rely on loop-carried variables introduced in the `with` clause. For example:

```
@compose {
  z = with x = x0 do op(x, y) while cond(x);
}
```

or

```
@compose {
  z = with x = x0 while cond(x) do op(x, y);
}
```

The same updating rule applies for loop-carried variables and scan outputs as in the case of pack based looping. The operator or block after the `while` keyword (the condition) can take any inputs (including the loop-carried variables), and **must** return a single `bool` tensor of singular shape (volume of 1), on which the continuation of the looping depends. The condition **may** itself be a loop-carried tensor of type `bool` and rank 0 (empty block yielding the tensor). This way it is possible to implement a loop with the body and condition computation collapsed into one operator or block, that yields the new value of the condition as well (in the corresponding position). The final value of the condition (which will be `false` if a loop count is not specified) can be omitted on the left-hand-side using the `~` symbol:

```
@compose {
  z, ~ = with x = x0, cond = True while cond do op(x, y);
}
```

In the above example, the operator op returns the new value of the condition as the second output.

Condition based looping can be combined with pack based looping by providing a for clause and/or an iteration count/index in the .. clause after the do keyword. In this case, the stopping condition is *either* the iteration index reaching the iteration count or the condition becoming false. Either a condition, a for clause or an explicit loop count **must** be present.

In essence, loops can have any combination loop-carried dependencies, loop local items of scan inputs, scan outputs, and conditions and iteration counts.

**Loop unrolling**

Although a compiler may choose to unroll loops on its own (when the loop count is a compile-time value), in certain situations the user might explicitly want to initiate the unrolling of loops. Even though the user could unroll some loops manually, in some cases, such as parametric sub-graphs, explicitly using loops is a better way of modeling the sub-graph (for example when a certain operation or group of operations is repeated a number of times, where the number of repeats is a compile-time parameter). For this reason, loops can explicitly be denoted as unrolled using the unroll keyword instead of the do keyword in some of the previous constructs. Naturally, loop unrolling only works with loops that have a compile-time loop count, hence unrolling cannot be used with condition based looping, only with scanning and folding loops iterating over packs, or loops that have explicit loop counts. In case of folding loops that declare loop carried dependencies in a with clause, unrolling results in a newly generated tensor for each loop carried dependency in each step of the loop (as opposed to the same tensor being updated in each step in case of a dynamic loop).

For example, the above example of add_all can be unrolled the following way (only difference is the unroll keyword instead of the do keyword):

```
operator add_all {
  @input {
    xs: real[n]..(k);
  }
  @output {
    y: real[n];
  }
  @compose {
    y = with sum: real[n] = 0.0 for x : xs unroll add(sum, x);
  }
}
```

## 2.11. Updating variables

Variable updates can be performed with the same syntax as composing computation of outputs, but in a separate block `@update`. Variables can only be assigned to in the `@update` block, and outputs can only be assigned to in the `@compose` block. Intermediate tensors can be defined in both blocks, and the intermediate tensors defined in the `@compose` block can be used in the `@update` block, as the `@update` block is executed after the `@compose` block. For example,

```
@attrib {
  update_rate: real;
}
@variable {
  w: real[s..];
}
@update {
  dw = ...;     # calculate delta
  w = mul_add(dw, update_rate, w);
}
```

## 2.12. Defining lower level computation

The computation of operators can be defined in a lower level as well, describing the computation by math formulas of tensor arithmetic in the `@lower` block. These formulas use compact indexing notation to describe parallel computation and rely on *local index symbols* introduced in a comma separated list after each formula. The index symbols are introduced in expressions of the form `id < bound`, where `id` is the identifier for the symbol, and `bound` is an expression defining its upper bound (exclusive), while the lower bound being implicitly 0 (inclusive). These indices denote *runtime* variables, as opposed to the compile-time ones used in variable initializer expressions.

For example, matrix multiplication can be described as follows:

```
operator matmul {
  @input {
    A: real[m,k];
    B: real[k,n];
  }
  @output {
    C: real[m,n];
  }
  @lower {
    C[i,j] += A[i,l] * B[l,j],
      i < m, l < k, j < n;
  }
}
```

The line `C[i,j] += A[i,l] * B[l,j]` basically describes a triple nested loop with variables `i`, `l`, `j`, and in the core of the loop the given value of tensor `C` is updated according to the formula, in this case becoming a sum of products `A[i,l] * B[l,j]`.

The basic building block of the notation is a *tensor access expression*; a tensor identifier followed by comma separated index expressions within `[]`. The access expression must have as many index dimensions as the rank of the tensor. The indices themselves may be packed (denoted by `..`), but the grouping need not be the same as in the shape declaration of the tensor.

The rules for writing such expressions are as follows:

- The left hand side **must** be an access expression for a single output tensor.
- The assignment operator **must** be one of `=`, `:=`, `+=`, `*=`, `&=`, `|=`, `<=`, `>=`, where `<=` and `>=` mean minimum and maximum, respectively, and operator `:=` means updating an initial tensor. For each output tensor, at most two expressions are allowed that have the given output on their left hand side: the first using the `=` operator for initialization of the computation, and the second using one of the rest of the operators for optionally updating/accumulating results. In case of operators `+=`, `*=`, `&=`, `|=`, the initialization can also be omitted, in which case the null element of the corresponding accumulation operator is used automatically for initialization (`0` for `+=`, `1` for `*=`, `true` for `&=` and `false` for `|=`). In case of operators `:=`, `<=`, `>=`, initialization **must** be present as they don't have mathematically well defined null elements.
- The right hand side may be a composition of unary and binary operators, the `?:` and `??` operators and built-in functions and cast expressions of tensor accesses or constant literals.
- Indexing of input and output tensor packs is also allowed.

For example, to elaborate the above example, let's provide an optional initial value to the result:

```
operator matmul {
  @input {
    A: real[m,k];
    B: real[k,n];
    Z: optional real[m,n];
  }
  @output {
    C: real[m,n];
  }
  @lower {
    C[i,j] = Z[i,j] ?? 0.0,
      i < m, j < n;
    C[i,j] += A[i,l] * B[l,j],
      i < m, l < k, j < n;
  }
}
```

In this case, if input `Z` is provided, it becomes the initial value of the summation, otherwise it is initialized to zeros.

If an index variable is used on the right-hand-side but not on the left-hand-side, it constitutes a reduction (for example `l` in the above example). If an index variable is used on the left-hand-side but not on the right-hand-side, it constitutes broadcasting (for example `i` and `j` on the first line above when `Z` is null). Expressions of optional type (for example `Z[i,j]` above due to null propagation from `Z`) must be resolved on the right-hand-side (for example using the `??` operator).

Index variables may also be used to address items of a tensor pack and this way define operators that process or produce tensor packs. For example, the previous example `add_to_all` can be written using lowering in the following way:

```
operator add_to_all {
  @input {
    xs: real[n,m]..(k);
    z: real[n,m];
  }
  @output {
    ys: real[n,m]..(k);
  }
  @lower {
    ys[ki][ni,mi] = xs[ki][ni,mi] + z[ni,mi],
      ki < k, ni < n, mi < m;
  }
}
```

**Packed assignment**

In case the assignment pertains to a complete pack (on the left-hand-side), the pack indexing may be omitted (from both sides). The above example could be written as

```
@lower {
  ys[ni,mi] = xs[ni,mi] + z[ni,mi],
    ni < n, mi < m;
}
```

However, there is a slight difference in the resulting semantics; in the latter case, the assignment to the pack is understood as one atomic operation inside the nested loop, while the previous writing had an explicit (outer) loop for iterating over the pack item by item. The difference may become significant when combined with conditions, as introduced in the next paragraph.

**Conditional assignment**

It is possible to guard the assignment with a condition, allowing it to be executed only if the condition evaluates to true.

The condition is written after the symbol `|`, after the enumeration of loop index symbols. The condition may depend on loop index variables or on tensor values, including the output being written in case the assignment is an update.

For example, an argmax search can be written as follows:

```
operator max_idx {
  @input {
    x: real[n];
  }
  @output {
    idx: int[];
  }
  @lower {
    idx[] = 0;
    idx[] := i,
      i < n | x[i,] > x[idx[],];
  }
}
```

In more complex cases, the output may be a pack and we wish to test and assign it in the conditional loop as one unit. Consider the 2 dimensional version of the above argmax example, where the indices have two components:

```
operator max_idx_2d {
  @input {
    x: real[m,n];
  }
  @output {
    idx: int[]..(2);
  }
  @lower {
    idx[] = 0;
    idx[] := [i,j],
      i < m, j < n | x[i,j] > x[idx[]..];
  }
}
```

Here, it is required to use the pack `idx` in the condition and update it in the assignment as one unit, otherwise, if a separate loop would be used to iterate over the pack the result would be wrong, as the condition test would be executed multiple times, with part of the result already overwritten.

**Loop unrolling**

In some cases, it is beneficial to unroll the loop that iterates through the pack of tensors. In this case, it is possible to do so by turning the index across the pack into an unrolled index, using the same syntax as in case of control flow loops, but only the part with the `unroll` keyword and specifying the loop index and loop count afterwards:

```
operator add_to_all {
  @input {
    xs: real[n,m]..(k);
    z: real[n,m];
  }
  @output {
    ys: real[n,m]..(k);
  }
  @lower {
    unroll..(ki -> k)
      ys[ki][ni,mi] = xs[ki][ni,mi] + z[ni,mi],
        ni < n, mi < m;
  }
}
```

In this case, the loop over the input/output pack of length `k` is unrolled, resulting in `k` independent tensor assignments, as if the the unrolled index would be the *outermost* loop, resulting in a sequence of `k` nested loops in runtime, one for each pack item.

**Intermediate values**

In some cases when the expression on the right-hand-side is complex and has repeated sub-expressions, it is beneficial to store an intermediate result and reuse it. Such an intermediate result is understood as a scalar value local to the nested loops of the lowering computation (not whole tensors). This can be done by splitting the expression into a sequence of comma separated sub-assignments (all using the same set of index variables, under the same set of index bounds), where only the last assignment has a tensor access on its left-hand-side, and all previous ones have a single intermediate identifier on their left-hand-side. For example, suppose we want to compute `x ** 2 + x ** 2 * sin(x)` element-wise, and want to avoid duplicating the sub-expression `x ** 2`:

```
@input {
  x: real[s..];
}
@output {
  y: real[s..];
}
@lower {
  z = x[i..] * x[i..],
  y[i] = z + z * `sin`(x[i..]),
    i < s;
}
```

In this example, the first assignment only stores `x ** 2` to the loop-local variable `z` (hence it is not terminated by a semi-colon), and the second assignment computes the final output. Both operate within the bounds `i < s`; the two lines form a single loop.

**Handling tensor boundaries**

In lowering expressions, simply respecting the bounds of individual tensor indices (the loop bounds) could still result in index expressions that themselves would index the respective tensors out of bounds. Such cases must be explicitly guarded against, and be marked by special syntax by placing such index expressions within `| |`. There are two ways to handle such out-of-bounds indices: *skipping* out-of-bounds entries, or *remapping* the index.

One canonical example is a sliding window filter, where the edges need special handling by skipping entries. For example, averaging (smoothing) an input `x` of length `s` with a window size `w`:

```
@attrib {
  w: int;
}
@input {
  x: real[s];
}
@output {
  y: real[s];
}
@lower {
  y[i] += x[|i + j - w/2|] / real(w),
    i < s, j < w;
}
```

Here, the index expression `i + j - w/2` may be out of bounds for input tensor `x` (on both the lower and the upper end), and hence must be guarded, marked by the `| |` around it. This tells the compiler to generate code to test the index for being in the valid range (implicitly defined by 0 on the lower end and the shape of the indexed tensor on the upper end), and skip the entries that would be out of bounds. In such a case, the complete (accumulating) expression is skipped, even if other tensors in the same expression would have valid indices.

In other cases, we might not want to skip such entries, but rather remap the indices. Such a case is denoted by the syntax `|i0 <> i1 : i2|` (where `i0`, `i1`, `i2` are all affine index expressions), meaning that the index to test against the tensor bounds is `i0`, and it is replaced by `i1` if `i0` would be less than 0, and replaced by `i2` if `i0` is larger or equal to the size of the indexed tensor in the given dimension. If index `i0` is within the bounds of the indexed tensor, then `i0` is used.

In the following example, the input is copied to the output, while duplicating the boundary values in the beginning and end of the tensor `n` times. The index expression `i - n` runs from `-n` (inclusive) to `s + n` (exclusive), but is clipped back to `0` and `s - 1` to address the boundary values:

```
@attrib {
  n: int;
}
@input {
  x: real[s];
}
@output {
  y: real[s + 2 * n];
}
@lower {
  y[i] = x[|i - n <> 0 : s - 1|],
    i < s + 2 * n;
}
```

## 2.13. Generics

Generic operators are those that have a `@dtype` block that introduces data-types used in the operator. A data-type definition consists of a name followed by an abstract base type (`type`, `num` or `arith`) to which the introduced type is restricted. The base type `type` allows any type, the base type `num` allows the types `int` and `real` to be substituted, while base type `arith` allows types `int`, `real` and `bool`. Furthermore, a default type may be given for invocations when the type cannot be deduced.

```
@dtype {
  T: num;
  S: type = real;
}
```

When a parameter is of type `type`, it can only participate in expressions that are valid for any type. When a parameter is of type `num`, it can only participate in expressions that are valid for any numeric type (for example arithmetic and comparison expressions). Parameters of type `arith` can participate in expressions valid for any type except for strings (such as casting).

Upon invocation of a generic operator, data-types that can be deduced from the inputs and attributes need not be specified explicitly. Data-types that cannot be deduced and do not have a default value must be specified upon invocation between `<>`:

```
x = op<real>(y);
```

The order of binding corresponds to the declaration order in the `@dtype` block.

## 2.14. Quantization

Quantization of tensors can be described in the `@quantize` block. The block consists of a list of pairs of previously declared tensor names and corresponding quantization algorithms (describes as an operator). The quantization algorithm itself is the binding of an operator with some attributes, but without an actual argument for invocation. The operator has to be such that it has exactly one input and one output, so that it could be invoked on the given tensor, and would return its quantized version. Quantization info can be provided for each input, output, intermediate tensor or variable of the operator or the graph. The names of the tensors may be scoped using (explicit or implicit) operator labels.

```
@compose {
  a = op1(x, y);
  b = op2(a, z);
}
@quantize {
  a: quant_op1{attr1=true};
  b: quant_op2{attr2=42};
}
```

## 2.15. Modules and imports

Operator definitions can be grouped into modules; a module is a single file. A module's (file) name automatically defines a namespace for operators. Modules can be imported into other modules via the `import` statement in the beginning of the file by providing the name of the module. Afterwards, the operators in the module can be referenced with qualified names, prefixed by the module name.

Module A:

```
operator op1 {
  # define operator
}
```

Module B:

```
import A;

operator op2 {
  @compose {
    x = A.op1(...);
  }
}
```

This way, different modules may have operators of the same name locally, but their qualified names differ. Module names are unique as a consequence of being files (whose names are unique). Modules **may** be placed into a model's root folder, or **may** be grouped into subfolders within a model folder, creating nested modules. Different subfolders **may** contain modules of the same name, but nested module names will be unique due to the folder structure. It is **recommended** to use a subfolder for custom operators to avoid future collision with standard module and operator names; standard modules are not contained in subfolders.

The module from which parsing starts is the main module, and **must** be named `main` in the model folder. It **must** contain at least one `graph` definition, while imported modules **must not** contain any `graph` definitions, only operator definitions. The main module **may** contain more than one `graph` definitions, such as different version of a model. The consuming software **may** have means to enumerate all graph definitions and pick one by the name for processing. In the absense of any (external) user input for which one to choose, the first one is considered to be the primary one, and is the one that **must** be processed.

## 2.16. Formal grammar definition

The below formal grammar definition uses the following notations

- regular identifiers (possibly including hyphens) are non-terminal symbols
- capital identifiers are terminal symbols
- bold strings between quotes (`"`) are keywords or operators
- items between square brackets (`[]`) are optional
- `item+` means one or more repetitions of an item
- `item*` means zero or more repetitions of an item
- `item+'` means one or more repetitions of comma separated items; shorthand for `item ("," item)*`
- `item*'` means zero or more repetitions of comma separated items; shorthand for `[ item ("," item)* ]`

The below grammar defines the allowed syntactic constructs in the broadest sense, that is, all possible combinations of constituents. However, optional elements may not be allowed in some contexts but allowed in others. Such constraints are not described here.

On the other hand, some expression constructs have a *type* property, and semantic validation checks the type matching of such expressions. Some of such type constraints (equality to a fixed type) are denoted here by indicating the type in subscript, for example $\texttt{expr}_{\texttt{int}}$ denotes and an expression of `int` type.

```
document ::= version extension* import* op-definition*
version ::= "version" INTEGER "." INTEGER ";"
extension ::= "extension" IDENTIFIER+ ";"
import ::= "import" IDENTIFIER+ ";"

op-definition ::= (["public"] "operator" | "graph") "{" definition-block+ "}"
definition-block ::= dtype-block
                   | attrib-block
                   | input-block
                   | output-block
                   | constant-block
                   | variable-block
                   | using-block
                   | assert-block
                   | compose-block
                   | lower-block
                   | update-block
                   | quantize-block

type-name ::= "type" | "arith" | "num" | "int" | "real" | "bool" | "str"
type-expr ::= type-name | IDENTIFIER
type-spec ::= ["optional"] type-expr [shape-spec] [pack-spec]
shape-spec ::= "[" (([".."] (rvalue-expr$_{int}$ | "~") ["|" rvalue-expr$_{int}$] [pack-spec])*' "]"
pack-spec ::= ".." ["(" rvalue-expr$_{int}$ ")"]

dtype-block ::= "@dtype" "{" dtype-definition* "}"
dtype-definition ::= IDENTIFIER ":" type-name ["=" type-name] ";"

attrib-block ::= "@attrib" "{" param-definition* "}"
input-block ::= "@input" "{" param-definition* "}"
output-block ::= "@output" "{" param-definition* "}"
constant-block ::= "@constant" "{" param-definition* "}"
variable-block ::= "@variable" "{" param-definition* "}"
param-definition ::= IDENTIFIER ":" type-spec ["=" rvalue-expr ("," loop-bound)*] ";"

using-block ::= "@using" "{" using* "}"
using ::= lvalue-expr "=" rvalue-expr ";"

assert-block ::= "@assert" "{" assertion* "}"
assertion ::= rvalue-expr$_{bool}$ [":" STRING ("," rvalue-expr)*] ";"

lower-block ::= "@lower" "{" lowering* "}"
lowering ::= ["unroll" iter-count]
             (loop-local-assignment ",")*
             tensor-access lowering-op rvalue-expr
             ("," loop-bound)* ["|" rvalue-expr$_{bool}$] ";"
loop-local-assignment ::= IDENTIFIER "=" rvalue-expr
loop-bound ::= IDENTIFIER "<" rvalue-expr$_{int}$
lowering-op ::= "=" | "+=" | "*=" | "&=" | "|=" | "<=" | ">=" | ":="

compose-block ::= "@compose" "{" component* "}"
update-block ::= "@update" "{" component* "}"
component ::= lvalue-expr+' "=" (invocation-expr | branch-expr | loop-expr) ";"
invocation-expr ::= invocation | rvalue-expr
branch-expr ::= "if" invocation-expr
                "then" invocation-expr
                ("elif" invocation-expr)*
                "else" invocation-expr
loop-expr ::= [with-clause] [for-clause] [while-clause]
              ("do" | "unroll") [iter-count] invocation [while-clause]

with-clause ::= "with" (IDENTIFIER "=" rvalue-expr)+'
for-clause ::= "for" (IDENTIFIER ":" rvalue-expr)+'
while-clause ::= "while" invocation-expr
iter-count ::= ".." "(" [IDENTIFIER "->"] rvalue-expr ")"
```

```
invocation ::= [IDENTIFIER ":"] qualified-identifier
               ["<" type-expr+' ">"]
               ["{" attribute-assignment+' "}"]
               "(" (rvalue-expr | "~")*' ")"
qualified-identifier ::= IDENTIFIER ("." IDENTIFIER)*
attribute-assignment ::= IDENTIFIER "=" rvalue-expr

lvalue-expr ::= lvalue-item
              | "[" lvalue-item+' "]"
lvalue-item ::= IDENTIFIER [":" type-expr shape-spec] [pack-spec]
              | "(" (IDENTIFIER [":" type-expr shape-spec])+' ")" [pack-spec]
              | "~"

rvalue-expr ::= IDENTIFIER [".shape" | ".rank" | ".size"]
              | literal
              | paren-expr
              | list-expr
              | subscript-expr
              | substitute-expr
              | unary-expr
              | binary-expr
              | select-expr
              | identity-expr
              | coalesce-expr
              | null-test-expr
              | contain-expr
              | fold-expr
              | cast-expr
              | builtin-expr
              | tensor-access

literal ::= INTEGER | REAL | STRING | "true" | "false" | "inf" | "pi"
paren-expr ::= "(" rvalue-expr ")"
list-expr ::= "[" list-item-expr*' "]"
list-item-expr ::= rvalue-expr | range-expr | expand-expr | zip-expr
range-expr ::= rvalue-expr ":" rvalue-expr [":" rvalue-expr]
expand-expr ::= rvalue-expr ".."
zip-expr ::= "(" rvalue-expr+' ")"
subscript-expr ::= rvalue-expr "[" rvalue-expr_int "]"
                 | rvalue-expr "[" [rvalue-expr_int] ":" [rvalue-expr_int]
                   [":" rvalue-expr_int] "]"
                 | rvalue-expr "[" rvalue-expr_bool "]"
substitute-expr ::= subscript-expr "<-" rvalue-expr
unary-expr ::= unary-op rvalue-expr
binary-expr ::= rvalue-expr binary-op rvalue-expr
select-expr ::= rvalue-expr "?" rvalue-expr [":" rvalue-expr]
coalesce-expr ::= rvalue-expr "??" rvalue-expr
identity-expr ::= rvalue-expr "is" rvalue-expr
contain-expr ::= rvalue-expr "in" rvalue-expr
fold-expr ::= rvalue-expr (fold-op ".." | cumlation-op "...")
```

```
cast-expr ::= type-expr "(" rvalue-expr ")"
builtin-expr ::= "`" builtin-func "`" "(" rvalue-expr ")"
bounded-expr ::= "|" rvalue-expr ["<>" rvalue-expr ":" rvalue-expr] "|"
tensor-access ::= IDENTIFIER ["[" rvalue-expr_int "]"]
                  "[" tensor-index_int*' | (tensor-index_int ",") "]"
tensor-index ::= rvalue-expr | bounded-expr

binary-op ::= binary-comparison-op
            | binary-arithmetic-op
            | binary-logical-op
binary-comparison-op ::= "<" | "<=" | ">" | ">=" | "==" | "!="
binary-arithmetic-op ::= "+" | "-" | "*" | "/" | "\" | "%" | "**" | "<<" | ">>"
binary-logical-op ::= "&&" | "||" | "^" | "=>"

unary-op ::= unary-arithmetic-op
           | unary-logical-op
           | null-test-op
unary-arithmetic-op ::= "+" | "-"
unary-logical-op ::= "!"
null-test-op ::= "?"

fold-op ::= fold-comparison-op
          | fold-arithmetic-op
          | fold-logical-op
fold-comparison-op ::= "<" | "<=" | ">" | ">=" | "==" | "!="
fold-arithmetic-op ::= "+" | "*" | "<<" | ">>"
fold-logical-op ::= "&&" | "||"
cumulation-op ::= fold-arithmetic-op
                | fold-logical-op

builtin-func ::= "abs" | "sign" | "sqrt"
               | "exp" | "log"
               | "sin" | "cos" | "tan"
               | "asin" | "acos" | "atan"
               | "sinh" | "cosh" | "tanh"
               | "asinh" | "acosh" | "atanh"
               | "ceil" | "floor" | "round"

quantize-block ::= "@quantize" "{" quantization* "}"
quantization ::= qualified-identifier ":" qualified-identifier
                     ["<" type-expr+' ">"]
                     ["{" attribute-assignment+' "}"] ";"
```

# Chapter 3. Graph composition

This section describes how an NNEF/SkriptND document may be processed as a first step towards producing an executable model out of it.

A compiler system **may** take an NNEF folder (a collection of files that import each other) and as a first step (after parsing and syntactic/semantic checking), **may** produce some intermediate representation of the model. In this section, *graph composition* means the process of generating a directed acyclic graph intermediate representation of tensors and operations (operator instances). The graph may potentially be hierarchical, containing references to subgraphs through special operations that take subgraph attributes. As an example, the resulting graph data structure **may** be built as follows (though does not need to be, details may vary across implementations):

- *Tensor* objects **may** store the following information: data-type, shape, optional constant value and quantization information (key-value pairs).
- *Operation* objects **may** store the following information: attributes (name-value pairs), generic types (name-type pairs), inputs and outputs (list of tensor references, each item possibly packed).
- *Graph* objects **may** maintain a list of all tensor objects (and dedicated ones for its inputs/outputs), and a list of operations, possibly in a topological order (valid execution order). Graphs may reference other sub-graphs through special control flow operations.

To compose a graph, the following pieces of information need to be supplied:

- The graph definition, potentially having graph level attributes and generic types
- If there are graph level attributes or generic types, each attribute and type must be assigned a concrete value
- Optionally, a list of operator names that should be / should not be decomposed during the composition process

Graph composition may proceed as follows:

- Given all the graph level attribute values (`@attrib` block), the sizes of all input tensors and model variables can be determined (`@input` and `@variable` blocks). This way, all input and variable objects can be constructed.
- Each operation is processed in order of definition in the `@compose` block. For each operation, the following must be performed in order:
    - The corresponding operator definition shall be retrieved
    - Attribute values shall be calculated for the invocation of the operation. Default values shall be substituted for missing attributes (using the `@attrib` block of the operator)
    - Input shapes shall be deduced using the input tensors in the invocation of the operation (using the `@input` block of the operator)
    - During the above two steps, generic data-types shall be deduced from the substituted attribute values and tensors. All types must become uniquely defined (`@dtype` block of the operator)

  - Intermediate compile-time values used in the operator definition shall be calculated (`@using` block of the operator)
  - Assert expressions shall be evaluated and checked for truth value (`@assert` block of the operator)
  - (Static) output shapes shall be calculated (`@output` block of the operator) and their data-types can be substituted from the previously deduced generic types if needed.
  - If the operation cannot or should not be decomposed, then output tensor objects shall be constructed and the operation object shall also be constructed given the attributes, data-types, references to inputs and outputs. If the operator contains lowering information (`@lower` block), the instantiation of the math formula can be attached to the operation instance for use in a later stage.
  - If the operator can and should be decomposed, each operation in its definition must be processed recursively (`@compose` block), generating intermediate tensor and operator objects.
- After all operations are processed, the graph object can be constructed from the list of all tensors and operations, and the list of graph level inputs and outputs.

A complete compiler system **may** then take this intermediate representation, optimize it and generate executable code to a target device. In order to do so, it can rely on the mathematical formulas for each operator (`@lower` block), or **may** generate (computationally equivalent) intrinsic code for any operator, potentially executing it on dedicated hardware.

## 3.1. Dynamic tensor shapes

As described by the syntax, the shape expressions for operator outputs specifies their shape in case the operation is static, or alternatively an upper bound the shapes in case the operator is dynamic (when its output shapes depend on the data itself). The output computation of such dynamic operators may or may not be possible to describe using the present syntax. On one hand, compound dynamic operators are possible to describe using dynamic loops or relying on other primitive dynamic operators. At the same time, most primitive dynamic operators are probably not possible to express using the affine index notation, and need to be left as intrinsics. Luckily, such cases seems rare, as dynamic operators can often be decomposed by relying on a few essential dynamic primitives.

In many model formats, operators may be *pseudo-dynamic*, in the sense that they have *tensor* inputs that almost always represent (expressions of) shapes of other tensors. Such expression of shapes would in most cases need not be dynamic, however, the formats cannot express such expressions in any other way, only as tensor computation. However, SkriptND *can* express such shape expressions without relying on tensor computation, hence it does not need to treat such operators as dynamic. These shape calculations can be interpreted both in compile-time and in run-time, as needed. When such expressions only depend on (ultimately input) tensor shapes and not tensor data (majpority of the cases), there is no need to treat such operators as dynamic, and shape inference can be calculated separately ahead of time of the actual tensor computation. Even in the case when such shape expressions are truly dynamic, they can be compiled to simple scalar code, there is no need to invoke massively parallel tensor machinery.

When the graph composer sees an operator whose output shape is marked as dynamic along a certain dimension, it can still propagate the shapes along the graph and compute upper bounds for all subsequent tensor shapes. Such an upper bound can be useful for example for pre-allocating buffers for execution. When the output tensor of an operation has dynamic shape (shape depends on the input tensor data), all the operations that use that output tensor as input also become dynamic in their output shapes as a consequence. Hence, this shape information needs to be propagated during *execution* of the model. More specifically, such shape inference calculations **must** be interleaved to some extent with the actual computation of the operations, recalculating the true dynamic shape of the output tensors based on the input shapes of such operations (that became dynamic due to previous dynamic operations). The expressions for such dynamic shape inference calculations are readily available from the syntax; they are the same expressions that are used for static shape inference (however, they need to be included in symbolic form in the graph IR, so that the code generator can compile them, but these can be represented as fairly simple scalar expressions). The graph composer can also note for the later code generation stage which shape calculations need to be generated into the execution code based on propagating the dynamic output markers starting from the originally dynamic operators.

# Chapter 4. Standard Library Operators

The Standard Library defines a set of often used operators from which more complex operators or complete graphs can be composed. The operators are grouped into various categories.

## 4.1. Data Layout Operators

The following operators are defined in the `layout` module.

### 4.1.1. Tensor Creation Operators

#### Constant of a given shape

```
operator constant {
    @dtype {
        T: type;
    }
    @attrib {
        shape: int..(d);
        value: T..(shape * ..);
    }
    @output {
        output: T[shape..];
    }
    @using {
        u = value := ..;
        prods = [1, shape[::-1]..] * ...;
        strides = prods[d-1::-1];
    }
    @lower {
        output[i..] = u ?? value[(i * strides) + ..],
            i < shape;
    }
}
```

#### Uniform value of a given shape

```
operator uniform {
    @dtype {
        T: type;
    }
    @attrib {
        shape: int..(d);
    }
    @input {
        value: T[];
    }
    @output {
        output: T[shape..];
    }
    @lower {
        output[i..] = value[],
            i < shape;
    }
}
```

**Range of values**

```
operator range {
    @dtype {
        T: num;
    }
    @attrib {
        first: T;
        last: T;
        stride: T = T(1);
    }
    @output {
        output: T[count];
    }
    @assert {
        stride != T(0);
        stride > T(0) => last >= first;
        stride < T(0) => first >= last;
    }
    @using {
        count = int((last - first) \ stride);
    }
    @lower {
        output[i,] = first + T(i) * stride,
            i < count;
    }
}
```

**Shape of a tensor as an integer tensor**

```
operator shape {
    @dtype {
        T: type;
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: int[d];
    }
    @lower {
        output[i,] = input.shape[i],
            i < d;
    }
}
```

### 4.1.2. Data View Operators

#### Data type casting

```
operator cast {
    @dtype {
        R: arith;
        T: arith;
    }
    @input {
        x: T[s..];
    }
    @output {
        y: R[s..];
    }
    @lower {
        y[i..] = R(x[i..]),
            i < s;
    }
}
```

#### Reshaping dimensions

```
operator reshape {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int = 0;      # dimension where reshaping starts
        rank: int = d;      # number of reshaped dimensions
        shape: int..(r);    # new shape for marked dimensions
    }
    @input {
        input: T^(input_rank)[s..(axis < 0 ? axis + input_rank : axis),t..(d)];
    }
    @output {
        output: T[s..,_shape..,t[rank:]..];
    }
    @using {
        input_volume = t[:rank] * ..;
        shape_volume = (shape is -1 ? 1 : shape) * ..;
        inferred_extent = shape_volume != 0 ? input_volume / shape_volume : 0;
        _shape = shape is -1 ? inferred_extent : shape;
    }
    @assert {
        axis >= -input.rank && axis < input.rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        rank >= 0; rank <= d:
            "'rank' must be between 0 (inclusive) and (input.rank - axis)
(exclusive)",
            input.rank, axis, rank;
        input_volume != 0 => shape > 0 || shape is -1:
            "'shape' must be positive or -1", shape;
        input_volume == 0 => shape >= 0:
            "'shape' must be non-negative", shape;
        ((shape is -1 ? 1 : 0) + ..) <= 1:
            "at most 1 item of 'shape' can be -1", shape;
        input_volume != 0 => input_volume % shape_volume == 0:
            "the product of input shape items (starting from 'axis', of length 'rank')
"
            "must be divisible by the product of the positive items of 'shape'",
            input.shape, axis, rank, shape;
        (_shape * ..) == input_volume:
            "the volume of 'shape' must be the same as implied by the "
            "input shape items (starting from 'axis', of length 'rank') ",
            'input-shape': [s..,t..], axis, rank, shape;
    }
    @compose {
        flattened = if rank == 1 then input else flatten{axis=axis, rank=rank}(input);
        output = if r == 1 then flattened else unflatten{axis=axis,
shape=_shape}(flattened);
    }
}
```

**Flattening dimensions**

```
operator flatten {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int = 0;      # dimension where flattening starts
        rank: int = d;      # number of flattened dimensions
    }
    @input {
        input: T^(input_rank)[s..(axis < 0 ? axis + input_rank : axis),t..(d)];
    }
    @output {
        output: T[s..,shape * ..,t[rank:d]..];
    }
    @assert {
        axis >= -input.rank && axis < input.rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        rank >= 0; rank <= d:
            "'rank' must be between 0 (inclusive) and input.rank (exclusive)",
            input.rank, rank;
    }
    @using {
        shape = t[:rank];
        prods = [1, shape[::-1]..] * ...;
        strides = prods[rank-1::-1];
    }
    @lower {
        output[si.., (i * strides) + .., ti..] = input[si.., i.., ti..],
            si < s, i < shape, ti < t[rank:d];
    }
}
```

```
operator unflatten {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;          # unflattened dimension
        shape: int..(d);    # new shape
    }
    @input {
        input: T^(input_rank)[s..(axis < 0 ? axis + input_rank : axis),z,t..];
    }
    @output {
        output: T[s..,shape..,t..];
    }
    @assert {
        axis >= -input.rank && axis < input.rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        shape > 0:
            "shape items must be positive", shape;
        (shape * ..) == z:
            "the volume of 'shape' must equal the input extent at dimension 'axis'",
            input.shape, axis, shape;
    }
    @using {
        prods = [1, shape[::-1]..] * ...;
        strides = prods[d-1::-1];
    }
    @lower {
        output[si.., i.., ti..] = input[si.., (i * strides) + .., ti..],
            si < s, i < shape, ti < t;
    }
}
```

### Squeezing dimensions

```
operator squeeze {
    @dtype {
        T: type;
    }
    @attrib {
        axes: int..(k);
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[s[kept_axes]..];
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
        zeros = [0 ..(d)];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        s[axes] == 1:
            "input.shape at 'axes' must be 1", input.shape, axes;
    }
    @lower {
        output[si..] = input[zeros[kept_axes] <- si..],
            si < s[kept_axes];
    }
}
```

```
operator unsqueeze {
    @dtype {
        T: type;
    }
    @attrib {
        axes: int..(k);
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[ones[comp_axes] <- s..];
    }
    @using {
        dims = [0:d+k];
        axes_mask = dims in (axes < 0 ? axes + (d + k) : axes);
        comp_axes..(d) = dims[!axes_mask];
        ones = [1 ..(d+k)];
        zeros = [0 ..(d+k)];
    }
    @assert {
        axes >= -(d + k) && axes < d + k:
            "'axes' must be between -output.rank (inclusive) and output.rank
(exclusive)",
            'output.rank': d+k, axes;
        axes != ..:
            "'axes' must be distinct", axes;
    }
    @lower {
        output[zeros[comp_axes] <- si..] = input[si..],
            si < s;
    }
}
```

### 4.1.3. Joining and Splitting Operators

#### Concatenating tensors along a dimension

```
operator concat {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
    }
    @input {
        inputs: T^(rank)[s..(axis < 0 ? axis + rank : axis),..z,t..]..(n);
    }
    @output {
        output: T[s..,u * n ?? (z + ..),t..];
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -input-rank (inclusive) and input-rank
(exclusive)",
            'input-rank': rank, axis;
    }
    @using {
        u = z := ..;
        sums = z + ...;
        offs = [0, sums..];
    }
    @lower {
        output[si..,(u * ni ?? offs[ni]) + zi,ti..] = inputs[ni][si..,zi,ti..],
            ni < n, si < s, zi < u ?? z[ni], ti < t;
    }
}
```

#### Splitting tensors along a dimension

```
operator split {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
        count: optional int;
        sizes: int..(n) = [(z / count)..(count)];
    }
    @input {
        input: T^(rank)[s..(axis < 0 ? axis + rank : axis),z,t..];
    }
    @output {
        outputs: T[s.., ..sizes, t..]..(n);
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        count >= 0:
            "'count' must be non-negative", count;
        n == count:
            "length of 'sizes' must equal 'count'", count, sizes;
        sizes >= 0:
            "'sizes' must be non-negative", sizes;
        (sizes + ..) == z:
            "sum of 'sizes' must equal input extent at dimension 'axis'",
            input.shape, axis, sizes;
    }
    @using {
        u = sizes := ..;
        sums = sizes + ...;
        offs = [0, sums..];
    }
    @lower {
        outputs[ni][si..,zi,ti..] = input[si..,(u * ni ?? offs[ni]) + zi,ti..],
            ni < n, si < s, zi < u ?? sizes[ni], ti < t;
    }
}
```

### Stacking tensors along a dimension

```
operator stack {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
        squeeze: bool = false;
    }
    @input {
        inputs: T^(rank)[s..(axis < 0 ? axis + rank : axis),1 ..(squeeze ? 1 :
0),t..]..(n);
    }
    @output {
        output: T[s..,n,t..];
    }
    @assert {
        axis >= -rank && axis <= rank:
            "'axis' must be between -input-rank (inclusive) and input-rank
(inclusive)",
            'input-rank': rank, axis;
    }
    @lower {
        output[si..,ni,ti..] = inputs[ni][si..,0 ..(squeeze ? 1 : 0),ti..],
            si < s, ti < t, ni < n;
    }
}
```

```
operator unstack {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
        squeeze: bool = true;
    }
    @input {
        input: T^(rank)[s..(axis < 0 ? axis + rank : axis),n,t..];
    }
    @output {
        outputs: T[s..,1 ..(!squeeze ? 1 : 0),t..]..(n);
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
    }
    @lower {
        outputs[ni][si..,0 ..(!squeeze ? 1 : 0),ti..] = input[si..,ni,ti..],
            si < s, ti < t, ni < n;
    }
}
```

### 4.1.4. Dimension Expanding and Shrinking Operators

#### Tiling along dimensions

```
operator tile {
    @dtype {
        T: type;
    }
    @attrib {
        axes: int..(k) = [0:d];     # effected axes
        repeats: int..(k);          # number of repeats for effected axes
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[s[axes] <- s[axes] * repeats..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != ..:
            "'axes' must be distinct", axes;
        repeats > 0:
            "'repeats' must be positive", repeats;
    }
    @lower {
        output[si[axes] <- ri * s[axes] + si[axes]..] = input[si..],
            si < s, ri < repeats;
    }
}
```

### Broadcasting along dimensions

```
operator broadcast {
    @dtype {
        T: type;
    }
    @attrib {
        axes: int..(k) = [0:d];     # effected axes
        shape: int..(k);            # new dimensions for effected axes
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[os..];
    }
    @using {
        os = s[axes] <- shape;
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != ..:
            "'axes' must be distinct", axes;
        shape > 0:
            "'shape' items must be positive", shape;
        s[axes] is 1 || s[axes] == shape:
            "input.shape must be 1 or equal to 'shape' at 'axes'",
            input.shape, axes, shape;
    }
    @lower {
        output[si..] = input[s is 1 ? 0 : si..],
            si < os;
    }
}
```

#### Slicing along dimensions

```
operator slice {
    @dtype {
        T: type;
    }
    @attrib {
        axes: int..(k) = [0:d];     # effected dimensions
        begin: int..(k);            # beginning positions for effected dimensions
        end: int..(k);              # ending positions for effected dimensions
        stride: int..(k) = 1;       # strides for effected dimensions
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[os..];
    }
    @using {
        _begin = ((begin < 0 ? begin + s[axes] : begin) >> -1) << s[axes];
        _end = ((end < 0 ? end + s[axes] : end) >> -1) << s[axes];
        os = s[axes] <- (_end - _begin) \ stride;
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != ..:
            "'axes' must be distinct", axes;
        stride != 0:
            "'stride' must be non-zero", stride;
        stride > 0 => _begin >= 0:
            "'begin' must be non-negative if 'stride' is positive";
        stride < 0 => _begin < s[axes]:
            "'begin' must be less than input.shape at 'axes' if stride is negative";
        stride > 0 => _end >= _begin:
            "'end' must be greater than or equal to begin if stride is positive";
        stride < 0 => _end <= _begin:
            "'end' must be less than or equal to begin if stride is negative";
    }
    @lower {
        output[si..] = input[si[axes] <- _begin + stride * si[axes]..],
            si < os;
    }
}
```

**Padding along dimensions**

```
operator pad {
    @dtype {
        T: type;
    }
    @attrib {
        axes: int..(k) = [0:d];
        padding: int..(2 * k);
        method: str = 'CONSTANT';
    }
    @input {
        input: T[s..(d)];
        value: optional T[];
    }
    @output {
        output: T[before + s + after..];
    }
    @using {
        methods = ['CONSTANT', 'REFLECT', 'REPLICATE', 'SYMMETRIC'];
        before = [0 ..(d)][axes] <- padding[:k];
        after = [0 ..(d)][axes] <- padding[k:];
    }
    @assert {
        method == methods || ..:
            "'method' must be one of {methods}", method;
        padding >= 0:
            "'padding' must be non-negative", padding;
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != ..:
            "'axes' must be distinct", axes;
    }
    @lower {
        zi = si - before,
        output[si..] = method == 'REFLECT' ? input[|zi <> -zi : 2 * (s - 1) - zi|..] :
                       method == 'REPLICATE' ? input[|zi <> 0 : s - 1|..] :
                       method == 'SYMMETRIC' ? input[|zi <> -zi - 1 : 2 * (s - 1) - zi
+ 1|..] :
                       (before + after == 0 || (zi >= 0 && zi < s)) && .. ?
input[zi..] :
                       (value[] ?? T()),
            si < before + s + after;
    }
}
```

### 4.1.5. Dimension Reordering Operators

**Transposing dimensions**

```
operator transpose {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int = 0;
        perm: int..(d) = [d-1:-1:-1];
    }
    @input {
        input: T^(rank)[z..(axis < 0 ? axis + rank : axis),s..(d)];
    }
    @output {
        output: T[z..,s[_perm - _axis]..];
    }
    @using {
        _axis = axis < 0 ? axis + input.rank : axis;
        _perm = perm < 0 ? perm + input.rank : perm;
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        _perm >= _axis && _perm < input.rank:
            "items in 'perm' must be between axis (inclusive) and input.rank
(exclusive) "
            "after correction for negative values",
            input.rank, axis, perm;
        perm != ..:
            "'perm' must be distinct", perm;
    }
    @lower {
        output[zi..,si[_perm - _axis]..] = input[zi..,si..],
            zi < z, si < s;
    }
}
```

### Merging space dimensions into / out of batch dimension

```
operator space_to_batch {
    @dtype {
        T: type;
    }
    @attrib {
        block_size: int..(d);
    }
    @input {
        input: T[b,c,s..(d)];
    }
    @output {
        output: T[block_size * .. * b,c,s / block_size..];
    }
    @assert {
        block_size > 0:
            "'block_size' must be positive", block_size;
        s % block_size == 0:
            "spatial size of input must be divisibe by 'block_size'",
            'size': s, block_size;
    }
    @compose {
        reshaped = reshape{axis=2, rank=d, shape=[(block_size,s /
block_size)..]}(input);
        transposed = transpose{perm=[2:2*d+2:2,0,1,3:2*d+2:2]}(reshaped);
        output = flatten{axis=0, rank=d+1}(transposed);
    }
}
```

```
operator batch_to_space {
    @dtype {
        T: type;
    }
    @attrib {
        block_size: int..(d);
    }
    @input {
        input: T[b,c,s..(d)];
    }
    @output {
        output: T[b / (block_size * ..),c,s * block_size..];
    }
    @assert {
        block_size > 0:
            "'block_size' must be positive", block_size;
        b % (block_size * ..) == 0:
            "batch size of input must be divisible by product of 'block_size'",
            'batch': b, block_size;
    }
    @using {
        range_d = [0:d];
    }
    @compose {
        reshaped = reshape{axis=0, rank=1, shape=[block_size..,b / (block_size *
..)]}(input);
        transposed = transpose{perm=[d,d+1,(range_d,range_d+d+2)..]}(reshaped);
        output = reshape{axis=2, rank=2*d, shape=[s * block_size..]}(transposed);
    }
}
```

### Shuffling channels

```
operator shuffle {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
        groups: int;
    }
    @input {
        input: T^(rank)[s..(axis < 0 ? axis + rank : axis),c,t..(k)];
    }
    @output {
        output: T[s..,c,t..];
    }
    @using {
        offs = axis < 0 ? axis + rank : axis;
    }
    @assert {
        axis >= -input.rank && axis < input.rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        groups > 0:
            "'groups' must be positive", groups;
        c % groups == 0:
            "input shape at 'axis' must be divisible by 'groups'",
            input.shape, axis, groups;
    }
    @compose {
        reshaped = reshape{axis=axis, rank=1, shape=[groups, c / groups]}(input);
        transposed = transpose{axis=axis,
perm=[offs+1,offs,offs+2:input.rank+1]}(reshaped);
        output = reshape{axis=axis, rank=2, shape=[c]}(transposed);
    }
}
```

### 4.1.6. Data Indexing Operators

**Finding indices where data is non-zero**

This operator produces an output of dynamic size depending on the actual data contained in the input.

It cannot be defined using affine index notation, hence it needs to be defined as an intrinsic operator in an actual compiler implementation.

```
operator nonzero {
    @dtype {
        T: arith;
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        index: int[d,~|(s * ..)];
    }
}
```

**Gathering data based on indices**

```
operator gather {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
    }
    @input {
        data: T^(rank)[s..(axis < 0 ? axis + rank : axis),z,t..];
        index: int[n..];
    }
    @output {
        output: T[s..,n..,t..];
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -data.rank (inclusive) and data.rank (exclusive)",
            data.rank, axis;
    }
    @lower {
        output[si..,ni..,ti..] = data[si..,index[ni..],ti..],
            si < s, ni < n, ti < t;
    }
}
```

```
operator gather_nd {
    @dtype {
        T: type;
    }
    @attrib {
        batch_dims: int = 0;
    }
    @input {
        data: T[b..(batch_dims),s..(d),t..];
        indices: int[b..(batch_dims),z..,d];
    }
    @output {
        output: T[b..,z..,t..];
    }
    @using {
        di = [0:d];
    }
    @assert {
        batch_dims >= 0: "'batch_dims' must be non-negative", batch_dims;
    }
    @lower {
        output[bi..,zi..,ti..] = data[bi..,indices[bi..,zi..,di]..,ti..],
            bi < b, zi < z, ti < t;
    }
}
```

#### Scattering data based on indices

```
operator scatter {
    @dtype {
        T: type;
    }
    @attrib {
        axis: int;
    }
    @input {
        data: T^(rank)[s..(axis),n,t..];
        indices: int[s..(axis),m,t..];
        updates: T[s..(axis),m,t..];
    }
    @output {
        output: T[s..,n,t..];
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -data.rank (inclusive) and data.rank (exclusive)",
            data.rank, axis;
    }
    @lower {
        output[si..,ni,ti..] = data[si..,ni,ti..],
            si < s, ni < n, ti < t;
        output[si..,indices[si..,mi,ti..],ti..] := updates[si..,mi,ti..],
            si < s, mi < m, ti < t;
    }
}
```

```
operator scatter_nd {
    @dtype {
        T: type;
    }
    @attrib {
        batch_dims: int = 0;
    }
    @input {
        data: T[b..(batch_dims),s..(d),t..];
        indices: int[b..(batch_dims),z..,d];
        updates: T[b..(batch_dims),z..,t..];
    }
    @output {
        output: T[b..,s..,t..];
    }
    @using {
        di = [0:d];
    }
    @assert {
        batch_dims >= 0: "'batch_dims' must be non-negative", batch_dims;
    }
    @lower {
        output[bi..,si..,ti..] = data[bi..,si..,ti..],
            bi < b, si < s, ti < t;
        output[bi..,indices[bi..,zi..,di]..,ti..] := updates[bi..,zi..,ti..],
            bi < b, zi < z, ti < t;
    }
}
```

## 4.2. Basic Math Operators

The following operators are defined in the `math` module.

### 4.2.1. Unary Element-wise Operators

#### Identity operator

```
operator iden {
    @dtype {
        T: type;
    }
    @input {
        x: T[s..];
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] = x[i..],
            i < s;
    }
}
```

**Element-wise negation**

```
operator neg {
    @dtype {
        T: num;
    }
    @input {
        x: T[s..];
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] = -x[i..],
            i < s;
    }
}
```

**Element-wise reciprocal**

```
operator rcp {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = 1.0 / x[i..],
            i < s;
    }
}
```

**Element-wise square**

```
operator sqr {
    @dtype {
        T: num;
    }
    @input {
        x: T[s..];
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] = x[i..] * x[i..],
            i < s;
    }
}
```

**Element-wise square root**

```
operator sqrt {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `sqrt`(x[i..]),
            i < s;
    }
}
```

**Element-wise reciprocal square**

```
operator rsqr {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = 1.0 / (x[i..] * x[i..]),
            i < s;
    }
}
```

**Element-wise reciprocal square root**

```
operator rsqrt {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = 1.0 / `sqrt`(x[i..]),
            i < s;
    }
}
```

**Element-wise exponential (natural base)**

```
operator exp {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `exp`(x[i..]),
            i < s;
    }
}
```

**Element-wise logarithm (natural base)**

```
operator log {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `log`(x[i..]),
            i < s;
    }
}
```

**Element-wise logarithm (base 2)**

```
operator log2 {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `log`(x[i..]) / `log`(2.0),
            i < s;
    }
}
```

**Element0wise trigonometric functions**

```
operator sin {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `sin`(x[i..]),
            i < s;
    }
}
```

```
operator cos {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `cos`(x[i..]),
            i < s;
    }
}
```

```
operator tan {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `tan`(x[i..]),
            i < s;
    }
}
```

```
operator sinh {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `sinh`(x[i..]),
            i < s;
    }
}
```

```
operator cosh {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `cosh`(x[i..]),
            i < s;
    }
}
```

```
operator tanh {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `tanh`(x[i..]),
            i < s;
    }
}
```

```
operator asin {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `asin`(x[i..]),
            i < s;
    }
}
```

```
operator acos {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `acos`(x[i..]),
            i < s;
    }
}
```

```
operator atan {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `atan`(x[i..]),
            i < s;
    }
}
```

```
operator asinh {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `asinh`(x[i..]),
            i < s;
    }
}
```

```
operator acosh {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `acosh`(x[i..]),
            i < s;
    }
}
```

```
operator atanh {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `atanh`(x[i..]),
            i < s;
    }
}
```

**Element-wise absolute value**

```
operator abs {
    @dtype {
        T: num;
    }
    @input {
        x: T[s..];
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] = x[i..] < T(0) ? -x[i..] : x[i..],
            i < s;
    }
}
```

**Element-wise sign**

```
operator sign {
    @dtype {
        T: num;
    }
    @input {
        x: T[s..];
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] = x[i..] > T(0) ? T(1) : x[i..] < T(0) ? T(-1) : T(0),
            i < s;
    }
}
```

**Element-wise logical not**

```
operator not {
    @input {
        x: bool[s..];
    }
    @output {
        y: bool[s..];
    }
    @lower {
        y[i..] = !x[i..],
            i < s;
    }
}
```

**Element-wise floor (rounding down)**

```
operator floor {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `floor`(x[i..]),
            i < s;
    }
}
```

**Element-wise ceil (rounding up)**

```
operator ceil {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `ceil`(x[i..]),
            i < s;
    }
}
```

**Element-wise rounding (to closest integer)**

```
operator round {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `round`(x[i..]),
            i < s;
    }
}
```

### 4.2.2. Binary Element-wise Operators

In case of binary element-wise operators, broadcasting must also be considered. When the ranks of the two arguments are the eqaul, broadcasting requires that the size of each dimension match or one of them is singular. If the ranks of the two arguments are not equal then one of is implicitly padded with singular dimensions either from left (first dimension) or right (last dimension).

By default, if the ranks of the two arguments are different, the arguments are aligned from the right, and the shorter one is padded with singular dimensions on the left for broadcasting. However, this behavior can be changed by aligning each of the arguments (with attributes `lhs_align` and `rhs_align`), padding with specified number of singular dimensions both from left (zero or positive alignment) or right (negative alignment). Note that the alignment of zero corresponds to complete left-alignment of the argument, while the complete right-alignment is achieved by not supplying the alignment attribute (default null value).

The whole alignment procedure can be summarized as follows. For each argument, calculate the *aligned rank* (sum of argument rank plus the absolute value of the alignment, if any), and pad both arguments with singular dimensions as specified by their alignment (left or right), if any. The result's rank will be the maximum of the two aligned ranks. Then for the argument that has lower aligned rank (shorter), align it (already padded) with the other one (already padded) from the left if the alignment (of the shorter) is positive or zero, or from the right if the alignment (of the shorter) is negative or not specified. Finally, pad the shorter with singular dimensions on the non-aligned end if necessary to make the two padded ranks equal.

All this alignment and padding computation is incorporated in the `@using` block of the binary operators. The final aligned and padded shapes are checked for being broadcast compatible in the `@assert` block.

**Element-wise addition**

```
operator add {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] + y[sy is 1 ? 0 :zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise subtraction**

```
operator sub {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] - y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise multiplication**

```
operator mul {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] * y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

### Element-wise division

```
operator div {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] / y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

### Element-wise modulo

```
operator mod {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] % y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise power**

```
operator pow {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] ** y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise minimum**

```
operator min {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] << y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

#### Element-wise maximum

```
operator max {
    @dtype {
        T: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] >> y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise less-than comparison**

```
operator lt {
    @dtype {
        X: num;
        Y: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: X[sx..(rx)];
        y: Y[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] < y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise greater-than comparison**

```
operator gt {
    @dtype {
        X: num;
        Y: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: X[sx..(rx)];
        y: Y[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] > y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise less-equal comparison**

```
operator le {
    @dtype {
        X: num;
        Y: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: X[sx..(rx)];
        y: Y[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] <= y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

### Element-wise greater-equal comparison

```
operator ge {
    @dtype {
        X: num;
        Y: num;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: X[sx..(rx)];
        y: Y[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] >= y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise equal**

```
operator eq {
    @dtype {
        X: type;
        Y: type;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: X[sx..(rx)];
        y: Y[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] == y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise not-equal comparison**

```
operator ne {
    @dtype {
        X: type;
        Y: type;
    }
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: X[sx..(rx)];
        y: Y[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] != y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

#### Element-wise logical and

```
operator and {
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: bool[sx..(rx)];
        y: bool[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] && y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

**Element-wise logical or**

```
operator or {
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: bool[sx..(rx)];
        y: bool[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] || y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

### Element-wise logical xor

```
operator xor {
    @attrib {
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        x: bool[sx..(rx)];
        y: bool[sy..(ry)];
    }
    @output {
        z: bool[sz..];
    }
    @using {
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ex is 1 ? ey : ex;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting ({ex} vs {ey} after
alignment)",
            'lhs.shape': x.shape, 'rhs.shape': y.shape, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = x[sx is 1 ? 0 : zi[px:px+rx]..] ^ y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

### 4.2.3. Ternary Select Operator

For the ternay (and higher arity operators in general), broadcasting works similarly to binary operators, but among all arguments.

**Element-wise conditional selection**

```
operator select {
    @dtype {
        T: type;
    }
    @attrib {
        cond_align: optional int;
        lhs_align: optional int;
        rhs_align: optional int;
    }
    @input {
        c: bool[sc..(rc)];
        x: T[sx..(rx)];
        y: T[sy..(ry)];
    }
    @output {
        z: T[sz..];
    }
    @using {
        qc = rc + `abs`(cond_align ?? 0);
        qx = rx + `abs`(lhs_align ?? 0);
        qy = ry + `abs`(rhs_align ?? 0);
        rz = qx >> qy >> qc;
        pc = (cond_align < 0 ? rz - qc : cond_align) ?? rz - rc;
        px = (lhs_align < 0 ? rz - qx : lhs_align) ?? rz - rx;
        py = (rhs_align < 0 ? rz - qy : rhs_align) ?? rz - ry;
        ec = [1 ..(pc), sc.., 1 ..(rz - rc - pc)];
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        sz = ec is 1 ? (ex is 1 ? ey : ex) : ec;
    }
    @assert {
        ex == ey || ex is 1 || ey is 1:
            "incompatible argument shapes for broadcasting "
            "({ex} vs {ey} after alignment for arguments 'x' and 'y')",
            'cond.shape': c.shape, 'lhs.shape': x.shape, 'rhs.shape': y.shape,
            cond_align, lhs_align, rhs_align;
        ex == ec || ex is 1 || ec is 1:
            "incompatible argument shapes for broadcasting "
            "({ex} vs {ec} after alignment for arguments 'x' and 'c')",
            'cond.shape': c.shape, 'lhs.shape': x.shape, 'rhs.shape': y.shape,
            cond_align, lhs_align, rhs_align;
        ey == ec || ey is 1 || ec is 1:
            "incompatible argument shapes for broadcasting "
            "({ey} vs {ec} after alignment for arguments 'y' and 'c')",
            'cond.shape': c.shape, 'lhs.shape': x.shape, 'rhs.shape': y.shape,
            cond_align, lhs_align, rhs_align;
    }
    @lower {
        z[zi..] = c[sc is 1 ? 0 : zi[pc:pc+rc]..] ?
                  x[sx is 1 ? 0 : zi[px:px+rx]..] :
                  y[sy is 1 ? 0 : zi[py:py+ry]..],
            zi < sz;
    }
}
```

### 4.2.4. Variadric n-ary Element-wise Operators

The following operators perform associative and commutative operations on arbitrary number of input tensors. They only accept inputs of the same shape.

**Sum of n tensors**

```
operator sum_n {
    @dtype {
        T: num;
    }
    @input {
        xs: T[s..]..(n);
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] += xs[k][i..],
            k < n, i < s;
    }
}
```

**Product of n tensors**

```
operator prod_n {
    @dtype {
        T: num;
    }
    @input {
        xs: T[s..]..(n);
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] *= xs[k][i..],
            k < n, i < s;
    }
}
```

**Minimum of n tensors**

```
operator min_n {
    @dtype {
        T: num;
    }
    @input {
        xs: T[s..]..(n);
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] <= xs[k][i..],
            k < n, i < s;
    }
}
```

### Maximum of n tensors

```
operator max_n {
    @dtype {
        T: num;
    }
    @input {
        xs: T[s..]..(n);
    }
    @output {
        y: T[s..];
    }
    @lower {
        y[i..] >= xs[k][i..],
            k < n, i < s;
    }
}
```

### Argmin across n tensors

```
operator argmin_n {
    @dtype {
        T: num;
    }
    @input {
        xs: T[s..]..(n);
    }
    @output {
        y: int[s..];
    }
    @lower {
        y[i..] = 0,
            i < s;
        y[i..] := k,
            k < n, i < s | xs[k][i..] < xs[y[i..]][i..];
    }
}
```

**Argmax across n tensors**

```
operator argmax_n {
    @dtype {
        T: num;
    }
    @input {
        xs: T[s..]..(n);
    }
    @output {
        y: int[s..];
    }
    @lower {
        y[i..] = 0,
            i < s;
        y[i..] := k,
            k < n, i < s | xs[k][i..] > xs[y[i..]][i..];
    }
}
```

**Logical or of n tensors**

```
operator any_n {
    @input {
        xs: bool[s..]..(n);
    }
    @output {
        y: bool[s..];
    }
    @lower {
        y[i..] |= xs[k][i..],
            k < n, i < s;
    }
}
```

**Logical and of n tensors**

```
operator all_n {
    @input {
        xs: bool[s..]..(n);
    }
    @output {
        y: bool[s..];
    }
    @lower {
        y[i..] &= xs[k][i..],
            k < n, i < s;
    }
}
```

### 4.2.5. Miscellaneous Element-wise Operators

**Element-wise weighted addition (a * x + b)**

```
operator axpb {
    @attrib {
        x_align: optional int;
        a_align: optional int;
        b_align: optional int;
    }
    @input {
        a: real[sa..(ra)];
        x: real[sx..(rx)];
        b: real[sb..(rb)];
    }
    @output {
        y: real[sy..];
    }
    @using {
        qx = rx + `abs`(x_align ?? 0);
        qa = ra + `abs`(a_align ?? 0);
        qb = rb + `abs`(b_align ?? 0);
        ry = qx >> qa >> qb;
        px = (x_align < 0 ? ry - qx : x_align) ?? ry - rx;
        pa = (a_align < 0 ? ry - qa : a_align) ?? ry - ra;
        pb = (b_align < 0 ? ry - qb : b_align) ?? ry - rb;
        ex = [1 ..(px), sx.., 1 ..(ry - rx - px)];
        ea = [1 ..(pa), sa.., 1 ..(ry - ra - pa)];
        eb = [1 ..(pb), sb.., 1 ..(ry - rb - pb)];
        sy = ex is 1 ? (ea is 1 ? eb : ea) : ex;
    }
    @assert {
        ea == ex || ea is 1 || ex is 1:
            "incompatible argument shapes for broadcasting "
            "({ea} vs {ex} for arguments 'a' and 'x')",
            a.shape, x.shape, b.shape, a_align, x_align, b_align;
        eb == ex || eb is 1 || ex is 1:
            "incompatible argument shapes for broadcasting "
            "({eb} vs {ex} for arguments 'b' and 'x')",
            a.shape, x.shape, b.shape, a_align, x_align, b_align;
        ea == eb || ea is 1 || eb is 1:
            "incompatible argument shapes for broadcasting "
            "({ea} vs {eb} for arguments 'a' and 'b')",
            a.shape, x.shape, b.shape, a_align, x_align, b_align;
    }
    @lower {
        y[i..] = a[sa is 1 ? 0 : i[pa:pa+ra]..] * x[sx is 1 ? 0 : i[px:px+rx]..] +
                 b[sb is 1 ? 0 : i[pb:pb+rb]..],
            i < sy;
    }
}
```

**Element-wise weighted addition (a * x + b * y)**

```
operator axpby {
    @attrib {
        x_align: optional int;
        y_align: optional int;
        a_align: optional int;
        b_align: optional int;
    }
    @input {
        a: real[sa..(ra)];
        x: real[sx..(rx)];
        b: real[sb..(rb)];
        y: real[sy..(ry)];
    }
    @output {
        z: real[sz..];
    }
    @using {
        qx = rx + `abs`(x_align ?? 0);
        qy = ry + `abs`(y_align ?? 0);
        qa = ra + `abs`(a_align ?? 0);
        qb = rb + `abs`(b_align ?? 0);
        rz = qx >> qy >> qa >> qb;
        px = (x_align < 0 ? rz - qx : x_align) ?? rz - rx;
        py = (y_align < 0 ? rz - qy : y_align) ?? rz - ry;
        pa = (a_align < 0 ? rz - qa : a_align) ?? rz - ra;
        pb = (b_align < 0 ? rz - qb : b_align) ?? rz - rb;
        ex = [1 ..(px), sx.., 1 ..(rz - rx - px)];
        ey = [1 ..(py), sy.., 1 ..(rz - ry - py)];
        ea = [1 ..(pa), sa.., 1 ..(rz - ra - pa)];
        eb = [1 ..(pb), sb.., 1 ..(rz - rb - pb)];
        sax = ea is 1 ? ex : ea;
        sby = eb is 1 ? ey : eb;
        sz = sax is 1 ? sby : sax;
    }
    @assert {
        ea == ex || ea is 1 || ex is 1:
            "incompatible argument shapes for broadcasting "
            "({ea} vs {ex} for arguments 'a' and 'x')",
            a.shape, x.shape, b.shape, y.shape, a_align, x_align, b_align, y_align;
        eb == ey || eb is 1 || ey is 1:
            "incompatible argument shapes for broadcasting "
            "({eb} vs {ey} for arguments 'b' and 'y')",
            a.shape, x.shape, b.shape, y.shape, a_align, x_align, b_align, y_align;
        sax == sby || sax is 1 || sby is 1:
            "incompatible argument shapes for broadcasting "
            "({sax} vs {sby} for 'ax' and 'by')",
            a.shape, x.shape, b.shape, y.shape, a_align, x_align, b_align, y_align;
    }
    @lower {
        z[i..] = a[sa is 1 ? 0 : i[pa:pa+ra]..] * x[sx is 1 ? 0 : i[px:px+rx]..] +
                 b[sb is 1 ? 0 : i[pb:pb+rb]..] * y[sy is 1 ? 0 : i[py:py+ry]..],
            i < sz;
    }
}
```

**Element-wise clamping**

```
operator clamp {
    @attrib {
        val_align: optional int;
        min_align: optional int;
        max_align: optional int;
    }
    @dtype {
        T: num;
    }
    @input {
        x: T[sx..(rx)];
        a: T[sa..(ra)];
        b: T[sb..(rb)];
    }
    @output {
        y: T[sy..];
    }
    @using {
        qx = rx + `abs`(val_align ?? 0);
        qa = ra + `abs`(min_align ?? 0);
        qb = rb + `abs`(max_align ?? 0);
        ry = qx >> qa >> qb;
        px = (val_align < 0 ? ry - qx : val_align) ?? ry - rx;
        pa = (min_align < 0 ? ry - qa : min_align) ?? ry - ra;
        pb = (max_align < 0 ? ry - qb : max_align) ?? ry - rb;
        ex = [1 ..(px), sx.., 1 ..(ry - rx - px)];
        ea = [1 ..(pa), sa.., 1 ..(ry - ra - pa)];
        eb = [1 ..(pb), sb.., 1 ..(ry - rb - pb)];
        sy = ex is 1 ? (ea is 1 ? eb : ea) : ex;
    }
    @assert {
        ex == ea || ex is 1 || ea is 1:
            "incompatible argument shapes for broadcasting "
            "({sx} vs {sa} for arguments 'a' and 'x')",
            'val.shape': x.shape, 'min.shape': a.shape, 'max.shape': b.shape,
            val_align, min_align, max_align;
        ex == eb || ex is 1 || eb is 1:
            "incompatible argument shapes for broadcasting "
            "({sx} vs {sb} for arguments 'b' and 'x')",
            'val.shape': x.shape, 'min.shape': a.shape, 'max.shape': b.shape,
            val_align, min_align, max_align;
        ea == eb || ea is 1 || eb is 1:
            "incompatible argument shapes for broadcasting "
            "({sa} vs {sb} for arguments 'a' and 'b')",
            'val.shape': x.shape, 'min.shape': a.shape, 'max.shape': b.shape,
            val_align, min_align, max_align;
    }
    @lower {
        xi = x[sx is 1 ? 0 : i[px:px+rx]..],
        ai = a[sa is 1 ? 0 : i[pa:pa+ra]..],
        bi = b[sb is 1 ? 0 : i[pb:pb+rb]..],
        y[i..] = xi < ai ? ai : xi > bi ? bi : xi,
            i < sy;
    }
}
```

### 4.2.6. Reduction Operators

Reduction operators reduce a set of values to a single value along one or more axes of a tensor (using various operations). The reduced dimensions may optionally be squeezed out.

**Minimum reduction**

```
operator min_reduce {
    @dtype {
        T: num;
    }
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @lower {
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] <= input[i..],
            i < s;
    }
}
```

**Maximum reduction**

```
operator max_reduce {
    @dtype {
        T: num;
    }
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @lower {
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] >= input[i..],
            i < s;
    }
}
```

### Sum reduction

```
operator sum_reduce {
    @dtype {
        T: num;
    }
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @lower {
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] += input[i..],
            i < s;
    }
}
```

**Product reduction**

```
operator prod_reduce {
    @dtype {
        T: num;
    }
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: T[s..(d)];
    }
    @output {
        output: T[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @lower {
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] *= input[i..],
            i < s;
    }
}
```

**Mean reduction**

```
operator mean_reduce {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @compose {
        sum = sum_reduce{axes=axes, squeeze=squeeze}(input);
        output = div(sum, real(s[axes] * ..));
    }
}
```

### Lp-norm reduction

```
operator lp_reduce {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
        p: real;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
        p > 0.0:
            "'p' must be positive", p;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @compose {
        abs = math.abs(input);
        pow = math.pow(abs, p);
        sum = sum_reduce{axes=axes, squeeze=squeeze}(pow);
        output = math.pow(sum, 1.0 / p);
    }
}
```

**Logical or reduction (any)**

```
operator any_reduce {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: bool[s..(d)];
    }
    @output {
        output: bool[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @lower {
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] |= input[i..],
            i < s;
    }
}
```

### Logical and reduction (all)

```
operator all_reduce {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: bool[s..(d)];
    }
    @output {
        output: bool[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
    }
    @lower {
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] &= input[i..],
            i < s;
    }
}
```

**Argmin reduction**

```
operator argmin {
    @attrib {
        axis: int;
        squeeze: bool = false;
    }
    @input {
        input: real^(rank)[s..(axis < 0 ? axis + rank : axis),z,t..];
    }
    @output {
        output: int[s..,1 ..(!squeeze),t..];
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        z > 0: "input shape along dimension 'axis' must be positive";
    }
    @lower {
        output[si..,0 ..(!squeeze),ti..] = 0,
            si < s, ti < t;
        output[si..,0 ..(!squeeze),ti..] := zi,
            si < s, zi < z, ti < t | input[si..,zi,ti..] < input[si..,output[si..,0
..(!squeeze),ti..],ti..];
    }
}
```

```
operator argmin_nd {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: int[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..]..(k);
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
        s[axes] > 0: "input shape along 'axes' must be positive";
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
        zeros = [0 ..(d)];
    }
    @lower {
        output[squeeze ? i : (zeros[kept_axes] <- i)..] = 0,
            i < s[kept_axes];
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] := i[axes],
            i < s | input[i..] < input[i[axes] <- output[squeeze ? i[kept_axes] :
(axes_mask ? 0 : i)..]..];
    }
}
```

### Argmax reduction

```
operator argmax {
    @attrib {
        axis: int;
        squeeze: bool = false;
    }
    @input {
        input: real^(rank)[s..(axis < 0 ? axis + rank : axis),z,t..];
    }
    @output {
        output: int[s..,1 ..(!squeeze),t..];
    }
    @assert {
        axis >= -rank && axis < rank:
            "'axis' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axis;
        z > 0: "input shape along dimension 'axis' must be positive";
    }
    @lower {
        output[si..,0 ..(!squeeze),ti..] = 0,
            si < s, ti < t;
        output[si..,0 ..(!squeeze),ti..] := zi,
            si < s, zi < z, ti < t | input[si..,zi,ti..] > input[si..,output[si..,0
..(!squeeze),ti..],ti..];
    }
}
```

```
operator argmax_nd {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: int[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..]..(k);
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
        s[axes] > 0: "input shape along 'axes' must be positive";
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
        zeros = [0 ..(d)];
    }
    @lower {
        output[squeeze ? i : (zeros[kept_axes] <- i)..] = 0,
            i < s[kept_axes];
        output[squeeze ? i[kept_axes] : (axes_mask ? 0 : i)..] := i[axes],
            i < s | input[i..] > input[i[axes] <- output[squeeze ? i[kept_axes] :
(axes_mask ? 0 : i)..]..];
    }
}
```

## Mean and variance reduction

```
operator moments {
    @attrib {
        axes: int..(k) = [0:d];
        squeeze: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        mean: real[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
        variance: real[squeeze ? s[kept_axes] : (axes_mask ? 1 : s)..];
    }
    @assert {
        axes >= -d && axes < d:
            "'axes' must be between -input.rank (inclusive) and input.rank
(exclusive)",
            input.rank, axes;
        axes != .. :
            "'axes' must be distinct", axes;
    }
    @using {
        dims = [0:d];
        axes_mask = dims in (axes < 0 ? axes + d : axes);
        kept_axes = dims[!axes_mask];
        squeeze_mean = squeeze && k == d;
    }
    @compose {
        _mean = mean_reduce{axes=axes, squeeze=squeeze_mean}(input);
        mean = if squeeze && !squeeze_mean then layout.squeeze{axes=axes}(_mean) else
_mean;
        diff = sub(input, _mean);
        square = sqr(diff);
        variance = mean_reduce{axes=axes, squeeze=squeeze}(square);
    }
}
```

## 4.3. Linear Algebra Operators

The following operators are defined in the `linalg` module.

**Dot product of vectors**

```
operator dot {
    @input {
        x: real[n];
        y: real[n];
        b: optional real[];
    }
    @output {
        z: real[];
    }
    @lower {
        z[] = b[] ?? 0.0;
        z[] += x[i,] * y[i,],
            i < n;
    }
}
```

**Matrix-vector multiplication**

```
operator matvec {
    @attrib {
        transA: bool = false;
    }
    @input {
        A: real[m,n];
        x: real[transA ? m : n];
        b: optional real[transA ? n : m];
    }
    @output {
        y: real[transA ? n : m];
    }
    @lower {
        y[i,] = b[i,] ?? 0.0,
            i < (transA ? n : m);
        y[i,] += A[transA ? j : i, transA ? i : j] * x[j,],
            i < (transA ? n : m), j < (transA ? m : n);
    }
}
```

**Matrix multiplication (possibly batched)**

```
operator matmul {
    @attrib {
        transA: bool = false;
        transB: bool = false;
    }
    @input {
        A: real[a..(da),mA,nA];
        B: real[b..(db),mB,nB];
        C: optional real[c..(dc), transA ? nA : mA, transB ? mB : nB];
    }
    @output {
        Z: real[z..,mZ,nZ];
    }
    @using {
        dz = da >> db;
        ea = [1 ..(dz - da), a..];
        eb = [1 ..(dz - db), b..];
        ec = [1 ..(dz - dc), c..];
        z = ea >> eb;
        mZ = transA ? nA : mA;
        nZ = transB ? mB : nB;
        kZ = transA ? mA : nA;
    }
    @assert {
        !transA && !transB => nA == mB:
            "last dimension of A must match first (non-batch) dimension of B "
            "when A and B are not transposed",
            'A.shape': [mA,nA], 'B.shape': [mB,nB];
         transA && !transB => mA == mB:
            "first (non-batch) dimension of A must match first (non-batch) dimension
of B "
            "when A is transposed and B is not transposed",
            'A.shape': [mA,nA], 'B.shape': [mB,nB];
        !transA && transB => nA == nB: "last dimension of A must match last dimension
of B "
            "when A is not transposed and B is transposed",
            'A.shape': [mA,nA], 'B.shape': [mB,nB];
         transA && transB => mA == nB: "first (non-batch) dimension of A must match "
            "last dimension of B when both A and B are transposed",
            'A.shape': [mA,nA], 'B.shape': [mB,nB];
        ea == eb || ea == 1 || eb == 1:
            "incompatible argument shapes of A and B for broadcasting batch dimensions
"
            "({ea} vs {eb} after alignment)",
            A.shape, B.shape;
        ec == z || ec == 1:
            "incompatible argument shape of C for broadcasting batch dimensions "
            "({ec} vs {z} after alignment)",
            C.shape, "result shape": [z..,mZ,nZ];
    }
    @lower {
        Z[l..,i,j] = C[c == 1 ? 0 : l[dz-dc:]..,i,j] ?? 0.0,
            l < z, i < mZ, j < nZ;
        Z[l..,i,j] += transA && transB ? A[a == 1 ? 0 : l[dz-da:]..,k,i] *
                                         B[b == 1 ? 0 : l[dz-db:]..,j,k] :
                                transA ? A[a == 1 ? 0 : l[dz-da:]..,k,i] *
                                         B[b == 1 ? 0 : l[dz-db:]..,k,j] :
                                transB ? A[a == 1 ? 0 : l[dz-da:]..,i,k] *
                                         B[b == 1 ? 0 : l[dz-db:]..,j,k] :
                                            A[a == 1 ? 0 : l[dz-da:]..,i,k] *
                                            B[b == 1 ? 0 : l[dz-db:]..,k,j],
            l < z, i < mZ, j < nZ, k < kZ;
    }
}
```

**Outer product of vectors**

```
operator outer {
    @input {
        x: real[sx..];
        y: real[sy..];
    }
    @output {
        z: real[sx..,sy..];
    }
    @lower {
        z[i..,j..] = x[i..] * y[j..],
            i < sx, j < sy;
    }
}
```

## 4.4. Neural Network Operators

The following operators are defined in the `nn` module.

### 4.4.1. Linear Operators

**Fully Connected Linear Operator**

```
operator linear {
    @input {
        input: real[b,c];
        filter: real[n,c];
        bias: optional real[n];
    }
    @output {
        output: real[b,n];
    }
    @lower {
        output[bi,ni] = bias[ni,] ?? 0.0,
            bi < b, ni < n;
        output[bi,ni] += input[bi,ci] * filter[ni,ci],
            bi < b, ci < c, ni < n;
    }
}
```

**Convolution**

```
operator conv {
    @attrib {
        stride: int..(d) = 1;
        dilation: int..(d) = 1;
        padding: optional int..(2 * d);
        padding_align: str = 'UPPER';
        ceil_mode: bool = false;
        groups: int = 1;
        data_format: str = 'NCX';
        filter_format: str = 'NCX';
    }
    @input {
        input: real[s..(d+2)];
        filter: real[fs..(d+2)];
        bias: optional real[bs];
    }
    @output {
        output: real[os..];
    }
    @using {
        b = data_format == 'XCN' ? s[-1] : s[0];
        ic = data_format == 'NCX' ? s[1] :
             data_format == 'NXC' ? s[-1] :
                                    s[-2];
        ix = data_format == 'NCX' ? s[2:] :
             data_format == 'NXC' ? s[1:-1] :
                                    s[:-2];
        n = filter_format == 'XCN' || filter_format == 'CXN' ? fs[-1] : fs[0];
        fc = filter_format == 'NCX' ? fs[1] :
             filter_format == 'NXC' ? fs[-1] :
             filter_format == 'XCN' ? fs[-2] :
                                      fs[0];
        fx = filter_format == 'NCX' ? fs[2:] :
             filter_format == 'NXC' ? fs[1:-1] :
             filter_format == 'XCN' ? fs[:-2] :
                                      fs[1:-1];
        fd = (fx - 1) * dilation + 1;
        paddings = padding[:d] + padding[d:] ??
                   ((ceil_mode ? ix \ stride : ix / stride) - 1) * stride + fd - ix;
        offset = padding[:d] ?? padding_align == 'UPPER' ? paddings / 2 : paddings \
2;
        ox = ceil_mode ? (ix + paddings - fd) \ stride + 1 :
                         (ix + paddings - fd) / stride + 1;
        os = data_format == 'NCX' ? [b,n,ox..] :
             data_format == 'NXC' ? [b,ox..,n] :
                                    [ox..,n,b];
        g = groups == 0 ? ic : groups;
        m = n / g;
        padding_aligns = ['LOWER', 'UPPER'];
        data_formats = ['NCX', 'NXC', 'XCN'];
        filter_formats = ['NCX', 'NXC', 'XCN', 'CXN'];
    }
    @assert {
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        groups >= 0: "'groups' must not be negative", groups;
        ix + paddings >= fd:
            "padded input-size must be greater than (dilated) filter-size",
            'input.size': ix, 'filter.size': fx, dilation, 'total-padding': paddings;
        n % g == 0:
            "output-channels must be divisible by groups",
            'output-channels': n, 'groups': g;
        ic == fc * g:
            "input-channels must equal filter-channels * groups",
            'input-channels': ic, 'filter-channels': fc, 'groups' : g;
        bs == n:
            "bias-size must equal output-channels of filter",
            "bias-size": bs, "output-channels": n;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
        data_format in data_formats:
            "'data_format' must be one of {data_formats}", data_format;
        filter_format in filter_formats:
            "'filter_format' must be one of {filter_formats}", filter_format;
    }
    @lower {
        oi = data_format == 'NCX' ? [bi,ni,i..] :
             data_format == 'NXC' ? [bi,i..,ni] :
                                    [i..,ni,bi],
        output[oi..] = bias[ni,] ?? 0.0,
            bi < b, ni < n, i < ox;

        oi = data_format == 'NCX' ? [bi,gi * m + mi,i..] :
             data_format == 'NXC' ? [bi,i..,gi * m + mi] :
                                    [i..,gi * m + mi,bi],
        ii = data_format == 'NCX' ? [bi,gi * fc + ci,|stride * i + dilation * j -
offset|..] :
             data_format == 'NXC' ? [bi,|stride * i + dilation * j - offset|..,gi * fc
+ ci] :
                                    [|stride * i + dilation * j - offset|..,gi * fc +
ci,bi],
        fi = filter_format == 'NCX' ? [gi * m + mi,ci,j..] :
             filter_format == 'NXC' ? [gi * m + mi,j..,ci] :
             filter_format == 'XCN' ? [j..,ci,gi * m + mi] :
                                      [ci,j..,gi * m + mi],
        output[oi..] += input[ii..] * filter[fi..],
            bi < b, gi < g, mi < m, ci < fc, i < ox, j < fx;
    }
}
```

**Deconvolution (transposed convolution)**

```
operator deconv {
    @attrib {
        stride: int..(d) = 1;
        dilation: int..(d) = 1;
        padding: optional int..(2 * d);
        padding_align: str = 'UPPER';
        output_size: optional int..(d);
        groups: int = 1;
        data_format: str = 'NCX';
        filter_format: str = 'NCX';
    }
    @input {
        input: real[s..(d+2)];
        filter: real[fs..(d+2)];
        bias: optional real[bs];
    }
    @output {
        output: real[os..];
    }
    @using {
        b = data_format == 'XCN' ? s[-1] : s[0];
        ic = data_format == 'NCX' ? s[1] :
             data_format == 'NXC' ? s[-1] : s[-2];
        ix = data_format == 'NCX' ? s[2:] :
             data_format == 'NXC' ? s[1:-1] : s[:-2];
        n = filter_format == 'XCN' || filter_format == 'CXN' ? fs[-1] : fs[0];
        fc = filter_format == 'NCX' ? fs[1] :
             filter_format == 'NXC' ? fs[-1] :
             filter_format == 'XCN' ? fs[-2] :
                                      fs[0];
        fx = filter_format == 'NCX' ? fs[2:] :
             filter_format == 'NXC' ? fs[1:-1] :
             filter_format == 'XCN' ? fs[:-2] :
                                      fs[1:-1];
        fd = (fx - 1) * dilation + 1;
        paddings = padding[:d] + padding[d:] ?? (ix - 1) * stride + fd - ix * stride;
        offset = padding[:d] ?? padding_align == 'UPPER' ? paddings / 2 : paddings \
2;
        ox = output_size ?? (ix - 1) * stride + fd - paddings;
        g = groups == 0 ? ic : groups;
        oc = fc * g;
        os = data_format == 'NCX' ? [b,oc,ox..] :
             data_format == 'NXC' ? [b,ox..,oc] :
                                    [ox..,oc,b];
        m = n / g;
        padding_aligns = ['LOWER', 'UPPER'];
        data_formats = ['NCX', 'NXC', 'XCN'];
        filter_formats = ['NCX', 'NXC', 'XCN', 'CXN'];
    }
    @assert {
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        groups >= 0: "'groups' must not be negative", groups;
        (output_size + paddings - fd) / stride + 1 == ix:
            "'output_size' must be compatible with what would be calculated from "
            "input-size, filter-size, 'stride', 'dilation' and 'padding'",
            output_size, 'output.size': ox, 'input.size': ix, 'filter.size': fx,
            stride, dilation, 'total-padding': paddings;
        n % g == 0:
            "input-channels must be divisible by 'groups'", 'input-channels': n,
'groups' : g;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
        data_format in data_formats:
            "'data_format' must be one of {data_formats}", data_format;
        filter_format in filter_formats:
            "'filter_format' must be one of {data_formats}", filter_format;
    }
    @lower {
        oi = data_format == 'NCX' ? [bi,ci,i..] :
             data_format == 'NXC' ? [bi,i..,ci] :
                                    [i..,ci,bi],
        output[oi..] = bias[ci,] ?? 0.0,
            bi < b, ci < oc, i < ox;

        oi = data_format == 'NCX' ? [bi,gi * fc + ci,|stride * i + dilation * j -
offset|..] :
             data_format == 'NXC' ? [bi,|stride * i + dilation * j - offset|..,gi * fc
+ ci] :
                                    [|stride * i + dilation * j - offset|..,gi * fc +
ci,bi],
        ii = data_format == 'NCX' ? [bi,gi * m + mi,i..] :
             data_format == 'NXC' ? [bi,i..,gi * m + mi] :
                                    [i..,gi * m + mi,bi],
        fi = filter_format == 'NCX' ? [gi * m + mi,ci,j..] :
             filter_format == 'NXC' ? [gi * m + mi,j..,ci] :
             filter_format == 'XCN' ? [j..,ci,gi * m + mi] :
                                      [ci,j..,gi * m + mi],
        output[oi..] += input[ii..] * filter[fi..],
            bi < b, ci < fc, gi < g, mi < m, i < ix, j < fx;
    }
}
```

### 4.4.2. Pooling Operators

```
operator max_pool {
    @attrib {
        axes: int..(k) = [2:d];
        size: int..(k);
        stride: int..(k) = 1;
        dilation: int..(k) = 1;
        padding: optional int..(2 * k);
        padding_align: str = 'UPPER';
        ceil_mode: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        sa = s[axes];
        fd = (size - 1) * dilation + 1;
        paddings = padding[:k] + padding[k:] ??
                   ((ceil_mode ? sa \ stride : sa / stride) - 1) * stride + fd - sa;
        offset = padding[:k] ?? padding_align == 'UPPER' ? paddings / 2 : paddings \
2;
        os = s[axes] <- (ceil_mode ? (sa + paddings - fd) \ stride + 1 :
                                       (sa + paddings - fd) / stride + 1);
        padding_aligns = ['LOWER', 'UPPER'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0: "'size' must be positive", size;
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        sa + paddings >= fd:
            "padded input-size must be greater than (dilated) filter-size",
            'input-size': sa, 'filter-size': size, dilation, 'total-padding':
paddings;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
    }
    @lower {
        output[i..] >= input[i[axes] <- |stride * i[axes] + dilation * j - offset|..],
            i < os, j < size;
    }
}
```

```
operator sum_pool {
    @attrib {
        axes: int..(k) = [2:d];
        size: int..(k);
        stride: int..(k) = 1;
        dilation: int..(k) = 1;
        padding: optional int..(2 * k);
        padding_align: str = 'UPPER';
        ceil_mode: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        sa = s[axes];
        fd = (size - 1) * dilation + 1;
        paddings = padding[:k] + padding[k:] ??
                   ((ceil_mode ? sa \ stride : sa / stride) - 1) * stride + fd - sa;
        offset = padding[:k] ?? padding_align == 'UPPER' ? paddings / 2 : paddings \
2;
        os = s[axes] <- (ceil_mode ? (sa + paddings - fd) \ stride + 1 :
                                     (sa + paddings - fd) / stride + 1);
        padding_aligns = ['LOWER', 'UPPER'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0: "'size' must be positive", size;
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        sa + paddings >= fd:
            "padded input-size must be greater than (dilated) filter-size",
            'input-size': sa, 'filter-size': size, dilation, 'total-padding':
paddings;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
    }
    @lower {
        output[i..] += input[i[axes] <- |stride * i[axes] + dilation * j - offset|..],
            i < os, j < size;
    }
}
```

```
operator avg_pool {
    @attrib {
        axes: int..(k) = [2:d];
        size: int..(k);
        stride: int..(k) = 1;
        dilation: int..(k) = 1;
        padding: optional int..(2 * k);
        padding_align: str = 'UPPER';
        ignore_border: bool = true;
        ceil_mode: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        sa = s[axes];
        fd = (size - 1) * dilation + 1;
        paddings = padding[:k] + padding[k:] ??
                   ((ceil_mode ? sa \ stride : sa / stride) - 1) * stride + fd - sa;
        os = s[axes] <- (ceil_mode ? (sa + paddings - fd) \ stride + 1 :
                                     (sa + paddings - fd) / stride + 1);
        padding_aligns = ['LOWER', 'UPPER'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0: "'size' must be positive", size;
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        sa + paddings >= fd:
            "padded input-size must be greater than (dilated) filter-size",
            'input-size': sa, 'filter-size': size, dilation, 'total-padding':
paddings;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
    }
    @constant {
        ones: real[[1 ..(d)][axes] <- sa..] = 1.0;
    }
    @compose {
        sum = sum_pool{size=size, stride=stride, dilation=dilation, padding=padding,
                        padding_align=padding_align, axes=axes,
ceil_mode=ceil_mode}(input);
        cnt = if ignore_border then
                  sum_pool{size=size, stride=stride, dilation=dilation,
padding=padding,
                           padding_align=padding_align, axes=axes,
ceil_mode=ceil_mode}(ones)
              else real(size * ..);
        output = math.div(sum, cnt);
    }
}
```

```
operator rms_pool {
    @attrib {
        axes: int..(k) = [2:d];
        size: int..(k);
        stride: int..(k) = 1;
        dilation: int..(k) = 1;
        padding: optional int..(2 * k);
        padding_align: str = 'UPPER';
        ignore_border: bool = true;
        ceil_mode: bool = false;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        sa = s[axes];
        fd = (size - 1) * dilation + 1;
        paddings = padding[:k] + padding[k:] ??
                   ((ceil_mode ? sa \ stride : sa / stride) - 1) * stride + fd - sa;
        os = s[axes] <- (ceil_mode ? (sa + paddings - fd) \ stride + 1 :
                                      (sa + paddings - fd) / stride + 1);
        padding_aligns = ['LOWER', 'UPPER'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0: "'size' must be positive", size;
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        sa + paddings >= fd:
            "padded input-size must be greater than (dilated) filter-size",
            'input-size': sa, 'filter-size': size, dilation, 'total-padding':
paddings;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
    }
    @compose {
        square = math.sqr(input);
        mean = avg_pool{size=size, stride=stride, dilation=dilation,
                        padding=padding, padding_align=padding_align,
                        axes=axes, ceil_mode=ceil_mode,
ignore_border=ignore_border}(square);
        output = math.sqrt(mean);
    }
}
```

```
operator lp_pool {
    @attrib {
        axes: int..(k) = [2:d];
        size: int..(k);
        stride: int..(k) = 1;
        dilation: int..(k) = 1;
        padding: optional int..(2 * k);
        padding_align: str = 'UPPER';
        ceil_mode: bool = false;
        p: real;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        sa = s[axes];
        fd = (size - 1) * dilation + 1;
        paddings = padding[:k] + padding[k:] ??
                   ((ceil_mode ? sa \ stride : sa / stride) - 1) * stride + fd - sa;
        os = s[axes] <- (ceil_mode ? (sa + paddings - fd) \ stride + 1 :
                                     (sa + paddings - fd) / stride + 1);
        padding_aligns = ['LOWER', 'UPPER'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0: "'size' must be positive", size;
        stride > 0: "'stride' must be positive", stride;
        dilation > 0: "'dilation' must be positive", dilation;
        sa + paddings >= fd:
            "padded input-size must be greater than (dilated) filter-size",
            'input-size': sa, 'filter-size': size, dilation, 'total-padding':
paddings;
        padding_align in padding_aligns:
            "'padding_align' must be one of {padding_aligns}", padding_align;
        p > 0.0: "'p' must be positive", p;
    }
    @compose {
        abs = math.abs(input);
        pow = math.pow(abs, p);
        sum = sum_pool{size=size, stride=stride, dilation=dilation,
                        padding=padding, padding_align=padding_align,
                        axes=axes, ceil_mode=ceil_mode}(pow);
        output = math.pow(sum, 1.0 / p);
    }
}
```

### 4.4.3. Activation Functions

```
operator relu {
    @attrib {
        alpha: optional real;
        max: optional real;
    }
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @assert {
        alpha >= 0.0 && alpha <= 1.0:
            "'alpha' must be between 0 and 1 (inclusive)'", alpha;
        max > 0.0:
            "'max' must be positive", max;
    }
    @lower {
        y[i..] = (x[i..] << max ?? x[i..]) >> (alpha * x[i..] ?? 0.0),
            i < s;
    }
}
```

```
operator prelu {
    @attrib {
        axis: int = 1;
    }
    @input {
        x: real[s..(d)];
        alpha: real[s[axis]];
    }
    @output {
        y: real[s..];
    }
    @assert {
        axis >= -d && axis < d:
            "axis must be between -x.rank (inclusive) and x.rank (exclusive)",
            x.rank, axis;
    }
    @lower {
        y[i..] = x[i..] < 0.0 ? alpha[i[axis],] * x[i..] : x[i..],
            i < s;
    }
}
```

```
operator thresholded_relu {
    @attrib {
        theta: real = 0.0;
    }
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @assert {
        theta >= 0.0: "'theta' must be positive", theta;
    }
    @lower {
        y[i..] = x[i..] > theta ? x[i..] : 0.0,
            i < s;
    }
}
```

```
operator elu {
    @attrib {
        alpha: real = 1.0;
    }
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = x[i..] < 0.0 ? alpha * (`exp`(x[i..]) - 1.0) : x[i..],
            i < s;
    }
}
```

```
operator selu {
    @attrib {
        alpha: real = 1.67326319;
        lambda: real = 1.05070102;
    }
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = lambda * (x[i..] < 0.0 ? alpha * (`exp`(x[i..]) - 1.0) : x[i..]),
            i < s;
    }
}
```

```
operator gelu {
    @attrib {
        approximate: optional str;
    }
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @using {
        approximations = ['TANH', 'SIGMOID'];
        approximation = approximate ?? '';
    }
    @assert {
        approximate in approximations:
            "'approximate' must be one of {approximations}", approximate;
    }
    @lower {
        y[i..] = approximation == 'TANH' ? 0.5 * x[i..] * (1.0 + `tanh`(`sqrt`(2.0 /
pi)
                                           * (x[i..] + 0.044715 * x[i..] ** 3.0))) :
                 approximation == 'SIGMOID' ? x[i..] / (1.0 + `exp`(-1.702 * x[i..]))
:
                                    0.5 * x[i..] * (1.0 + `erf`(x[i..] /
`sqrt`(2.0))),
            i < s;
    }
}
```

```
operator silu {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = x[i..] / (1.0 + `exp`(-x[i..])),
            i < s;
    }
}
```

```
operator sigmoid {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = 1.0 / (1.0 + `exp`(-x[i..])),
            i < s;
    }
}
```

```
operator softplus {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `log`(`exp`(x[i..]) + 1.0),
            i < s;
    }
}
```

```
operator erf {
    @input {
        x: real[s..];
    }
    @output {
        y: real[s..];
    }
    @lower {
        y[i..] = `erf`(x[i..]),
            i < s;
    }
}
```

### 4.4.4. Normalization Operators

```
operator batch_norm {
    @attrib {
        epsilon: real = 1e-5;
        channel_axis: int = 1;
    }
    @input {
        input: real[s..(d)];
        mean: real[c];
        variance: real[c];
        bias: optional real[c];
        scale: optional real[c];
    }
    @output {
        output: real[s..];
    }
    @using {
        align = channel_axis < 0 ? d + channel_axis : channel_axis;
    }
    @assert {
        epsilon >= 0.0: "'epsilon' must be non-negative", epsilon;
        s[channel_axis] == c: "input shape at channel axis must match size of mean and
variance",
                              'input-shape': s, channel_axis, 'channels': c;
    }
    @compose {
        centered = math.sub{rhs_align=align}(input, mean);
        stabilized = math.add(variance, epsilon);
        std = math.sqrt(stabilized);
        normalized = math.div{rhs_align=align}(centered, std);
        scaled = if ?scale then math.mul{rhs_align=align}(normalized, scale) else
normalized;
        output = if ?bias then math.add{rhs_align=align}(scaled, bias) else scaled;
    }
}
```

```
operator mean_variance_norm {
    @attrib {
        axes: int.. = [2:d+2];
        epsilon: real = 1e-5;
    }
    @input {
        input: real[b,c,s..(d)];
        bias: optional real[c];
        scale: optional real[c];
    }
    @output {
        output: real[b,c,s..];
    }
    @assert {
        axes >= -(d + 2) && axes < d + 2:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        epsilon >= 0.0:
            "'epsilon' must be non-negative", epsilon;
    }
    @compose {
        mean, variance = moments: math.moments{axes=axes}(input);
        centered = math.sub(input, mean);
        stabilized = math.add(variance, epsilon);
        sigma = math.sqrt(stabilized);
        normalized = math.div(centered, sigma);
        scaled = if ?scale then math.mul{rhs_align=1}(normalized, scale) else
normalized;
        output = if ?bias then math.add{rhs_align=1}(scaled, bias) else scaled;
    }
}
```

```
operator local_response_norm {
    @attrib {
        axes: int..(k) = [1];
        size: int..(k);
        alpha: real = 1.0;
        beta: real = 0.5;
        bias: real = 1.0;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s..];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        alpha >= 0.0: "'alpha' must be non-negative", alpha;
        beta >= 0.0: "'beta' must be non-negative", beta;
        bias >= 0.0: "'bias' must be non-negative", bias;
    }
    @compose {
        square = math.sqr(input);
        mean = avg_pool{axes=axes, size=size, padding=[(size - 1) / 2 .., (size - 1) \
2 ..],
                        ignore_border=false}(square);
        scaled = math.mul(mean, alpha);
        biased = math.add(scaled, bias);
        powered = math.pow(biased, beta);
        output = math.div(input, powered);
    }
}
```

```
operator l1_norm {
    @attrib {
        axes: int..;
        bias: optional real;
        epsilon: optional real;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s..];
    }
    @assert {
        bias >= 0.0: "'bias' must be non-negative", bias;
        epsilon >= 0.0: "'epsilon' must be non-negative", epsilon;
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
    }
    @compose {
        abs = math.abs(input);
        sum = math.sum_reduce{axes=axes}(abs);
        biased = if ?bias then math.add(sum, bias) else sum;
        capped = if ?epsilon then math.max(biased, epsilon) else biased;
        output = math.div(input, capped);
    }
}
```

```
operator l2_norm {
    @attrib {
        axes: int..;
        bias: optional real;
        epsilon: optional real;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s..];
    }
    @assert {
        bias >= 0.0: "'bias' must be non-negative", bias;
        epsilon >= 0.0: "'epsilon' must be non-negative", epsilon;
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
    }
    @compose {
        square = math.sqr(input);
        sum = math.sum_reduce{axes=axes}(square);
        sigma = math.sqrt(sum);
        biased = if ?bias then math.add(sigma, bias) else sigma;
        capped = if ?epsilon then math.max(biased, epsilon) else biased;
        output = math.div(input, capped);
    }
}
```

```
operator softmax {
    @attrib {
        axes: int.. = [1];
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s..];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
    }
    @compose {
        max = math.max_reduce{axes=axes}(input);
        shifted = math.sub(input, max);
        exped = math.exp(shifted);
        sum = math.sum_reduce{axes=axes}(exped);
        output = math.div(exped, sum);
    }
}
```

### 4.4.5. Recurrent Operators

**LSTM**

This is a module-private helper operator to describe a single step of an LSTM.

```
operator _lstm_step {
    @input {
        h0: real[b,n];
        c0: real[b,n];
        x: real[b,c];
        W: real[4*n,c];
        R: real[4*n,n];
        B: real[4*n];
    }
    @output {
        h1: real[b,n];
        c1: real[b,n];
    }
    @compose {
        y = linear(x, W, B);
        z = linear(h0, R);
        s = math.add(y, z);
        [i, f, g, o] = layout.split{axis=1, count=4}(s);
        sf = sigmoid(f);
        si = sigmoid(i);
        so = sigmoid(o);
        tg = math.tanh(g);
        c1 = math.axpby(sf, c0, si, tg);
        tc = math.tanh(c1);
        h1 = math.mul(so, tc);
    }
}
```

This is a module-private helper operator to describe an LSTM iteration where each batch item has the same number of steps (which may be a static or dynamic count).

```
operator _lstm_loop {
    @input {
        X: real[s,b,c];
        W: real[4*n,c];
        R: real[4*n,n];
        B: real[4*n];
        h0: real[b,n];
        c0: real[b,n];
        steps: optional int[];
    }
    @output {
        Y: real[~|s,b,n];
        hN: real[b,n];
        cN: real[b,n];
    }
    @compose {
        Xs..(s) = layout.unstack{axis=0}(X);
        hN, cN, hs = with hi = h0, ci = c0 for Xi : Xs do..(steps) {
            h1, c1 = step: _lstm_step(hi, ci, Xi, W, R, B);
            yield h1, c1, h1;
        };
        Y = layout.stack{axis=0}(hs);
    }
}
```

This is a module-private helper operator to describe an LSTM iteration where each batch item may have different number of steps (dynamic counts).

```
operator _jagged_lstm_loops {
    @input {
        X: real[s,b,c];
        W: real[4*n,c];
        R: real[4*n,n];
        B: real[4*n];
        h0: real[b,n];
        c0: real[b,n];
        steps: int[b];
    }
    @output {
        Y: real[~|s,b,n];
        hN: real[b,n];
        cN: real[b,n];
    }
    @compose {
        counts..(b) = layout.unstack{axis=0}(steps);
        Xs: real[s,1,c]..(b) = layout.unstack{axis=1, squeeze=false}(X);
        h0s: real[1,n]..(b) = layout.unstack{axis=0, squeeze=false}(h0);
        c0s: real[1,n]..(b) = layout.unstack{axis=0, squeeze=false}(c0);
        Ys..(b), hNs..(b), cNs..(b) = for Xi : Xs, h0i : h0s, c0i : c0s, cnt : counts
do..(b)
                                        item: _lstm_loop(Xi, W, R, B, h0i, c0i, cnt);
        Y = layout.stack{axis=1, squeeze=true}(Ys);
        hN = layout.stack{axis=0, squeeze=true}(hNs);
        cN = layout.stack{axis=0, squeeze=true}(cNs);
    }
}
```

This is the public LSTM operator. The step count may be static (given by the first dimension of the input shape, same for all batch items) or dynamic if provided (potentially different for each batch item).

```
operator lstm {
    @input {
        X: real[s,b,c];
        W: real[4*n,c];
        R: real[4*n,n];
        B: real[4*n];
        h0: real[b,n] = 0.0;
        c0: real[b,n] = 0.0;
        steps: optional int[b];
    }
    @output {
        Y: real[~|s,b,n];
        hN: real[b,n];
        cN: real[b,n];
    }
    @compose {
        Y, hN, cN = if ?steps then
                        _jagged_lstm_loops(X, W, R, B, h0, c0, steps)
                    else
                        _lstm_loop(X, W, R, B, h0, c0);
    }
}
```

## 4.5. Image Processing Operators

The following operators are defined in the `image` module.

### 4.5.1. Image Up/down-Sampling Operators

```
operator nearest_downsample {
    @attrib {
        axes: int..(k) = [0:d];
        factor: int..(k);
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        os = s[axes] <- s[axes] / factor;
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        factor > 0:
            "'factor' must be positive", factor;
        s[axes] % factor == 0:
            "input.shape at 'axes' must be divisible by 'factor'",
            input.shape, axes, factor;
    }
    @lower {
        output[si..] = input[si[axes] <- si[axes] * factor..],
            si < os;
    }
}
```

```
operator nearest_upsample {
    @attrib {
        axes: int..(k) = [0:d];
        factor: int..(k);
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        os = s[axes] <- s[axes] * factor;
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        factor > 0:
            "'factor' must be positive", factor;
    }
    @lower {
        output[si[axes] <- si[axes] * factor + fi..] = input[si..],
            si < s, fi < factor;
    }
}
```

```
operator area_downsample {
    @attrib {
        axes: int..(k) = [0:d];
        factor: int..(k);
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        os = s[axes] <- s[axes] / factor;
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        factor > 0:
            "'factor' must be positive", factor;
        s[axes] % factor == 0:
            "input.shape at 'axes' must be divisible by 'factor'",
            input.shape, axes, factor;
    }
    @compose {
        sum = nn.sum_pool{size=factor, stride=factor, axes=axes}(input);
        output = math.div(sum, real(factor * ..));
    }
}
```

```
operator linear_upsample {
    @attrib {
        axes: int..(k) = [0:d];
        factor: int..(k);
        symmetric: bool = true;
        replicate_border: bool = true;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s[axes] <- (factor * s[axes])..];
    }
    @using {
        fs = symmetric ? 2 * factor - factor % 2 : 2 * factor - 1;
        padding = symmetric ? (replicate_border ? factor + factor / 2 : factor / 2) :
factor - 1;
        offset = symmetric && factor % 2 == 0 ? 0.5 : 1.0;
        axes_mask = [0:d] in axes;
        c = int(symmetric);
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        factor > 0:
            "'factor' must be positive", factor;
    }
    @constant {
        filter: real[fs..] = (1.0 - `abs`(real(i - factor) + offset) / real(factor)) *
..,
                             i < fs;
    }
    @lower {
        output[i[axes] <- |factor * i[axes] + j - padding|..]
            += input[replicate_border && axes_mask ? |i - c <> 0 : s - 1| : i..] *
filter[j..],
            i < replicate_border && axes_mask ? s + c + 1 : s, j < fs;
    }
}
```

### 4.5.2. Image Resize/Rescale Operators

The following operators cannot be expressed using affine indexing, hence they need to be defined as intrinsic operators in an actual compiler implementation.

```
operator nearest_resize {
    @attrib {
        axes: int..(k) = [0:d];
        size: int..(k);
        coordinate_transform: str = 'SYMMETRIC';
        rounding_method: str = 'ROUND_PREFER_FLOOR';
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @using {
        coordinate_transforms = ['SYMMETRIC', 'ASYMMETRIC', 'ALIGNED'];
        rounding_methods = ['FLOOR', 'CEIL', 'ROUND_PREFER_FLOOR',
'ROUND_PREFER_CEIL'];
        scale = coordinate_transform != 'ALIGNED' ? real(s[axes]) / real(size) :
                        (size <= 1 ? 0.0 : real(s[axes] - 1) / real(size - 1));
        offset = coordinate_transform == 'SYMMETRIC' ? 0.5 * scale - 0.5 : 0.0;
        os = s[axes] <- size;
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0:
            "'size' must be positive", size;
        coordinate_transform in coordinate_transforms:
            "'coordinate_transform' must be one of {coordinate_transforms}",
            coordinate_transform;
        rounding_method in rounding_methods:
            "'rounding_method' must be one of {rounding_methods}",
            rounding_method;
    }
    @lower {
        r = real(i[axes]) * scale + offset,
        output[i..] = input[i[axes] <- (int(rounding_method == 'FLOOR' ? `floor`(r) :
                                    rounding_method == 'CEIL' ? `ceil`(r) :
                                    rounding_method == 'ROUND_PREFER_FLOOR' ?
                                       `ceil`(r - 0.5) : `floor`(r + 0.5))
                                << (s[axes] - 1))..],
            i < os;
    }
}
```

```
operator linear_resize {
    @attrib {
        axes: int..(k) = [0:d];
        size: int..(k);
        antialias: bool = true;
        coordinate_transform: str = 'SYMMETRIC';
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[os..];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0:
            "'size' must be positive", size;
        coordinate_transform in coordinate_transforms:
            "'coordinate_transform' must be one of {coordinate_transforms}",
            coordinate_transform;
    }
    @using {
        coordinate_transforms = ['SYMMETRIC', 'ASYMMETRIC', 'ALIGNED'];
        scale = coordinate_transform != 'ALIGNED' ? real(s[axes]) / real(size) :
                        (size <= 1 ? 0.0 : real(s[axes] - 1) / real(size - 1));
        offset = coordinate_transform == 'SYMMETRIC' ? 0.5 * scale - 0.5 : 0.0;
        os = s[axes] <- size;
        w = [2 ..(k)];
    }
    @lower {
        r = real(i[axes]) * scale + offset,
        output[i..] += `abs`(real(1 - j) - `frac`(`abs`(r))) * ..
                     * input[i[axes] <- |int(`floor`(r)) + j <> 0 : s[axes] - 1|..],
            i < os, j < w;
    }
}
```

```
operator cubic_resize {
    @attrib {
        axes: int..(k) = [0:d];
        size: int..(k);
        antialias: bool = true;
        coeff_a: real = -0.75;
        coordinate_transform: str = 'SYMMETRIC';
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s[axes] <- size..];
    }
    @using {
        coordinate_transforms = ['SYMMETRIC', 'ASYMMETRIC', 'ALIGNED'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0:
            "'size' must be positive", size;
        coordinate_transform in coordinate_transforms:
            "'coordinate_transform' must be one of {coordinate_transforms}",
            coordinate_transform;
    }
}
```

```
operator resize {
    @attrib {
        axes: int..(k) = [0:d];
        size: int..(k);
        mode: str = 'NEAREST';
        coordinate_transform: str = 'SYMMETRIC';
        rounding_method: str = 'ROUND_PREFER_FLOOR';
        antialias: bool = true;
        cubic_coeff_a: real = -0.75;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s[axes] <- size..];
    }
    @using {
        modes = ['NEAREST', 'LINEAR', 'CUBIC'];
        coordinate_transforms = ['SYMMETRIC', 'ASYMMETRIC', 'ALIGNED'];
        rounding_methods = ['FLOOR', 'CEIL', 'ROUND_PREFER_FLOOR',
'ROUND_PREFER_CEIL'];
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0:
            "'size' must be positive", size;
        mode in modes:
            "'mode' must be one of {modes}", mode;
        coordinate_transform in coordinate_transforms:
            "'coordinate_transform' must be one of {coordinate_transforms}",
            coordinate_transform;
        rounding_method in rounding_methods:
            "'rounding_method' must be one of {rounding_methods}",
            rounding_method;
    }
    @compose {
        output = if mode == 'NEAREST' then
                     nearest_resize{axes=axes, size=size,
                                    coordinate_transform=coordinate_transform,
                                    rounding_method=rounding_method}(input)
                 elif mode == 'LINEAR' then
                     linear_resize{axes=axes, size=size, antialias=antialias,
                                   coordinate_transform=coordinate_transform}(input)
                 else
                     cubic_resize{axes=axes, size=size, antialias=antialias,
                                  coordinate_transform=coordinate_transform,
                                  coeff_a=cubic_coeff_a}(input);

    }
}
```

```
operator rescale {
    @attrib {
        axes: int..(k) = [0:d];
        factor: real..(k);
        mode: str = 'NEAREST';
        coordinate_transform: str = 'SYMMETRIC';
        rounding_method: str = 'ROUND_PREFER_FLOOR';
        antialias: bool = true;
        cubic_coeff_a: real = -0.75;
    }
    @input {
        input: real[s..(d)];
    }
    @output {
        output: real[s[axes] <- size..];
    }
    @using {
        modes = ['NEAREST', 'LINEAR', 'CUBIC'];
        coordinate_transforms = ['SYMMETRIC', 'ASYMMETRIC', 'ALIGNED'];
        rounding_methods = ['FLOOR', 'CEIL', 'ROUND_PREFER_FLOOR',
'ROUND_PREFER_CEIL'];
        size = int(`round`(real(s[axes]) * factor));
    }
    @assert {
        axes >= -d && axes < d:
            "axes must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axes;
        axes != ..:
            "axes must be distinct", axes;
        size > 0:
            "'size' must be positive", size;
        mode in modes:
            "'mode' must be one of {modes}", mode;
        coordinate_transform in coordinate_transforms:
            "'coordinate_transform' must be one of {coordinate_transforms}",
            coordinate_transform;
        rounding_method in rounding_methods:
            "'rounding_method' must be one of {rounding_methods}",
            rounding_method;
    }
    @compose {
        output = resize{axes=axes, size=size, mode=mode,
rounding_method=rounding_method,
                        coordinate_transform=coordinate_transform,
antialias=antialias,
                        cubic_coeff_a=cubic_coeff_a}(input);
    }
}
```

### 4.5.3. ROI Operators

The following operators cannot be expressed using affine indexing, hence they need to be defined as intrinsic operators in an actual compiler implementation.

```
operator max_roi_pool {
    @attrib {
        output_size: int..(d);
    }
    @input {
        input: real[b,c,s..(d)];
        rois: real[m,2 * d];
        index: int[m];
    }
    @output {
        output: real[b,c,output_size..];
    }
}
```

```
operator avg_roi_pool {
    @attrib {
        output_size: int..(d);
    }
    @input {
        input: real[b,c,s..(d)];
        rois: real[m,2 * d];
        index: int[m];
    }
    @output {
        output: real[b,c,output_size..];
    }
}
```

## 4.6. Quatization Operators

The following operators are defined in the `quant` module.

**Linear Quantization**

```
operator zero_point_linear_quantize {
    @attrib {
        zero_point: int..(s[channel_axis]);
        scale: real..(s[channel_axis]);
        bits: int;
        signed: bool = true;
        symmetric: bool = false;
        channel_axis: int = 1;
    }
    @input {
        input: real[s..];
    }
    @output {
        output: real[s..];
    }
    @using {
        r = signed ? 2 ** (bits - 1) - 1 : 2 ** bits - 1;
        m = !signed ? 0 : symmetric ? -r : -r - 1;
        uz = zero_point := ..;
        us = scale := ..;
    }
    @assert {
        scale > 0.0: "scale must be positive", scale;
        bits > 0: "bits must be positive", bits;
    }
    @lower {
        zp = uz ?? zero_point[i[channel_axis]],
        sc = us ?? scale[i[channel_axis]],
        quantized = zp + int(`round`(input[i..] / sc)) >> m << r,
        output[i..] = real(quantized - zp) * sc,
            i < s;
    }
}
```

```
operator min_max_linear_quantize {
    @attrib {
        min: real..(s[channel_axis]);
        max: real..(s[channel_axis]);
        bits: int;
        signed: bool = true;
        symmetric: bool = false;
        channel_axis: int = 1;
    }
    @input {
        input: real[s..];
    }
    @output {
        output: real[s..];
    }
    @using {
        r = real(2 ** bits - 1 - int(signed && symmetric));
        p = real(signed ? 2 ** (bits - 1) - int(symmetric) : 0);
        umin = min := ..;
        umax = max := ..;
    }
    @assert {
        bits > 0: "bits must be positive", bits;
    }
    @lower {
        mn = umin ?? min[i[channel_axis]],
        mx = umax ?? max[i[channel_axis]],
        sc = mx - mn,
        quantized = `round`(((input[i..] >> mn << mx) - mn) / sc * r) - p,
        output[i..] = real((quantized + p) / r * sc + mn),
            i < s;
    }
}
```

## 4.7. Algorithmic operators

The following operators are defined in the `algo` module.

The below operators are inherently sequential and cannot be expressed with massively parallelizable closed form math formulae, hence they need to be defined as intrinsic operators in an actual compiler implementation.

**Top-k element selection**

```
operator top_k {
    @attrib {
        k: int;
        axis: int = 0;
        largest: bool = true;
        sorted: bool = true;
    }
    @input {
        input: real^(rank)[s..(axis < 0 ? axis + rank : axis),m,t..];
    }
    @output {
        values: real[s..,k << m,t..];
        indices: int[s..,k << m,t..];
    }
    @assert {
        axis >= -rank && axis < rank:
            "axis must be between -input.rank (inclusive) and input.rank (exclusive)",
            input.rank, axis;
        k > 0: "'k' must be positive", k;
    }
}
```

**Non-maximum suppression**

This operator returns a dynamic number of outputs (batch dimension) depending on the input data.

```
operator nonmax_suppress {
    @attrib {
        box_format: str = "CORNERS";
        max_outputs_per_class: int = n;
        iou_threshold: real = 0.0;
        score_threshold: optional real;
    }
    @input {
        boxes: real[b,n,4];
        scores: real[b,c,n];
    }
    @output {
        indices: int[~|b * c * max_outputs_per_class,3];
    }
    @using {
        bbox_formats = ["CORNERS", "CENTER"];
    }
    @assert {
        box_format in bbox_formats:
            "'box_format' must be one of {bbox_formats}", box_format;
        iou_threshold >= 0.0 && iou_threshold <= 1.0:
            "'iou_threshold' must be between 0 and 1 (inclusive)", iou_threshold;
        max_outputs_per_class >= 0:
            "'max_outputs_per_class' must be non-negative", max_outputs_per_class;
    }
}
```