



# Neural Network Exchange Format

The Khronos NNEF Working Group

Version 1.0.1, Revision 5, 2019-05-10

# Table of Contents

1. Introduction	2
1.1. What is NNEF	2
1.1.1. The Exporter's view of NNEF	2
1.1.2. The Importer's view of NNEF	3
1.1.3. The Application Programmer's view of NNEF	3
1.1.4. What NNEF is not	3
1.2. Specification Terminology	3
2. Fundamentals	5
2.1. Computational Graphs	5
2.2. Description of Data	6
2.3. Description of Operations	6
2.4. Overview of Graph Description and Usage	7
2.4.1. Graph Compilation and Execution	8
2.5. Glossary of Terms	8
3. Formal Description	11
3.1. Lexical Elements	11
3.2. Syntax	12
3.2.1. Graph Definition	12
3.2.2. Fragment Definition	13
3.2.3. Operator Expressions	14
3.2.4. The Whole Document	16
3.3. Semantics	17
3.3.1. Type System and Type Checking	18
3.3.2. Graph and Fragment Definition	20
3.3.3. Building Expressions	23
3.3.4. Exported Identifiers	27
4. Operations	29
4.1. Tensor Introducing Operations	29
4.1.1. External Data Sources	29
4.1.2. Constants	30
4.1.3. Variables	30
4.2. Element-wise Operations	31
4.2.1. Unary Operations	31
4.2.2. Binary Operations	32
4.2.3. Select Operation	33
4.2.4. Simplifier Operations	34
4.3. Sliding-Window Operations	36
4.3.1. Convolution and Deconvolution	38

4.3.2. Box Filter	41
4.3.3. Index Based Sampling	42
4.3.4. Up and Down-Sampling	44
4.4. Reduce Operations	47
4.5. Tensor Shape Operations	49
4.5.1. Reshaping	49
4.5.2. Transposing	50
4.5.3. Splitting and Concatenation	50
4.5.4. Slicing	52
4.6. Region-of-Interest Operations	53
4.6.1. RoI Pooling	53
4.6.2. RoI Align	54
4.7. Matrix Multiplication	55
4.8. Variable Updates	56
4.9. Compound Operations	57
4.9.1. Activation Functions	57
4.9.2. Linear Operations	58
4.9.3. Pooling Operations	59
4.9.4. Normalization Operations	61
4.9.5. Quantization Operations	63
4.9.6. Miscellaneous Operations	63
5. Storing Network Data	65
5.1. Container Organization	65
5.2. Tensor File Format	65
5.3. Quantization File Format	67
6. Document Validity	68
7. Quantization	69
7.1. Incorporating Quantization Info	69
7.2. Dynamic Quantization	71
Appendix A: Grammar Definitions	73
A.1. Flat Grammar	73
A.2. Compositional Grammar	73
Appendix B: Example Export	76
B.1. AlexNet	76
Appendix C: Credits	78
Appendix D: List of Changes	79

## Copyright 2017 The Khronos Group Inc.

This specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos.

This specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at [www.khronos.org/files/member\\_agreement.pdf](http://www.khronos.org/files/member_agreement.pdf). Khronos grants a conditional copyright license to use and reproduce the unmodified specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms. Parties desiring to implement the specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos IP Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this specification; see <https://www.khronos.org/adopters>.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Vulkan is a registered trademark and Khronos, OpenXR, SPIR, SPIR-V, SYCL, WebGL, WebCL, OpenVX, OpenVG, EGL, COLLADA, glTF, NNEF, OpenKODE, OpenKCAM, StreamInput, OpenWF, OpenSL ES, OpenMAX, OpenMAX AL, OpenMAX IL, OpenMAX DL, OpenML and DevU are trademarks of The Khronos Group Inc. ASTC is a trademark of ARM Holdings PLC, OpenCL is a trademark of Apple Inc. and OpenGL and OpenML are registered trademarks and the OpenGL ES and OpenGL SC logos are trademarks of Silicon Graphics International used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

# Chapter 1. Introduction

This chapter is Informative except for the section on Terminology.

This document, referred to as the `0NNEF Specification0` or just the `0Specification0` hereafter, describes the Neural Network Exchange Format: 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 operations and terminology.

## 1.1. What is NNEF

NNEF is a data format for exchanging information about (trained) neural networks. Exchanging such information 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, as different accelerators are compatible with different frameworks. The goal of NNEF is to provide a standard platform for connecting accelerated neural network execution engines and available deep learning tools. Ideally, neural networks trained in deep learning frameworks would be exported to NNEF, and neural network accelerator libraries could consume it without worrying about compatibility with all deep learning frameworks.

NNEF aims to encapsulate two key aspects of neural networks: *network structure* and *network data*. To describe network structure in a flexible way, NNEF introduces a simple syntax similar to existing scripting languages, along with a set of standardized operations to express common neural network architectures. The syntax is designed to be both human readable and editable, and also easy to parse by consumer libraries. The network data, which form parameters of the structure, is stored in a simple format that supports flexible production and consumption of networks. NNEF can be thought of as a simple language with which a neural network accelerator can be programmed, while being independent of many details of the training and inference process, such as how the network is fed with data, or the data representations and algorithms of the underlying hardware.

Although the main focus of NNEF is to be a central chain in the pipeline from deep learning frameworks to neural network 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. Therefore, producers and consumers of NNEF may be various, however, two important sub-categories are exporters and importers, explained below.

### 1.1.1. The Exporter's view of NNEF

For an exporter of NNEF, such as a deep learning framework, NNEF is a format to which its internal network representation can be converted, and afterwards all accelerator libraries that are able to consume NNEF will be able to execute the trained network (if the network matches its capabilities). The task of an exporter is to map its operations and data representation to the operations and data representation of NNEF, given that this mapping is possible.

NNEF also aims to be 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, especially 'local' ones that require a locally concentrated subset of the input data to compute outputs.

### 1.1.2. The Importer's view of NNEF

The importer of NNEF, such as a neural network accelerator library (and its underlying hardware), is able to import an NNEF document, and compile it to its internal representation ready for execution. This compilation may happen offline or online. During offline compilation, NNEF may be converted to an optimized, hardware specific representation format, which may be saved and later be quickly loaded for execution. During online compilation, the conversion and optimization may happen without saving it into a hardware specific format, but immediately executing the converted network.

NNEF collects operations into groups to indicate relatedness of operations. This may serve as a hint or guideline for hardware implementations, as related operations may require similar hardware capabilities.

### 1.1.3. The Application Programmer's view of NNEF

For an application programmer, NNEF is a standardized way to store and transfer neural networks. Given a neural network in NNEF format, and a driver or library that is able to import it, the application programmer need not worry about where the network came from or what kind of underlying hardware will execute it, as long as it has the capabilities to do so. The application programmer may query the driver of the underlying hardware whether it is capable of executing the given network.

### 1.1.4. 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 what it means to correctly execute a neural network described in NNEF. Although it does 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.

Libraries 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. Specification Terminology

The key words must, required, should, recommend, may, and optional in this document are to be

interpreted as described in RFC 2119:

<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 of the specification. When followed by not (‘must not’), the phrase means that the definition is an absolute prohibition of the specification.

#### should

When used alone, this word means that there may exist valid reasons in particular circumstances to ignore a particular item, 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 adjective optional, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation that does not include a particular option must be prepared to interoperate with another implementation, which does include the option, though perhaps with reduced functionality. In the same vein an implementation that does include a particular option must be prepared to interoperate with another implementation, which does not include the option (except, of course, for the feature the option provides).

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

#### can

This word means that the particular behavior described is a valid choice for an application, and is never used to refer to implementation behavior.

#### cannot

This word means that the particular behavior described is not achievable by an application.



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 format is capable of expressing, but that the consequences of doing so are undefined and potentially unrecoverable for an implementation.

# Chapter 2. Fundamentals

This chapter introduces fundamental concepts including computational graphs, operations and tensors, NNEF syntax and data description. It provides a framework for interpreting more specific descriptions of operations and network architectures in the remainder of the Specification.

## 2.1. Computational Graphs

A neural network can be described by a computational graph. The computational graph is a directed graph that has two types of nodes: *data* nodes and *operation* nodes. A directed edge from a data node to an operation node means that the operation takes the data as its input, while a directed edge from an operation node to a data node means that the operation produces the data as its output. Edges from data node to data node or from operation node to operation node are not allowed.

As an example, a simple multi-layer feedforward network can be described by a linear graph, starting from an input data node, where each layer corresponds to an operation node producing a new intermediate data node, while taking the previous (intermediate) data as input, finally producing some output data.

Data nodes represent multi-dimensional arrays (such as vectors, matrices, or arrays of higher dimension), called *tensors*. The computation starts from data nodes that represent externally provided data or constant or variable tensors internal to the graph. In order to make the description uniform, such data nodes are the results of special [tensor introducing operations](#). Thus, the whole computational graph is described by a set of operations, interconnected by data nodes.

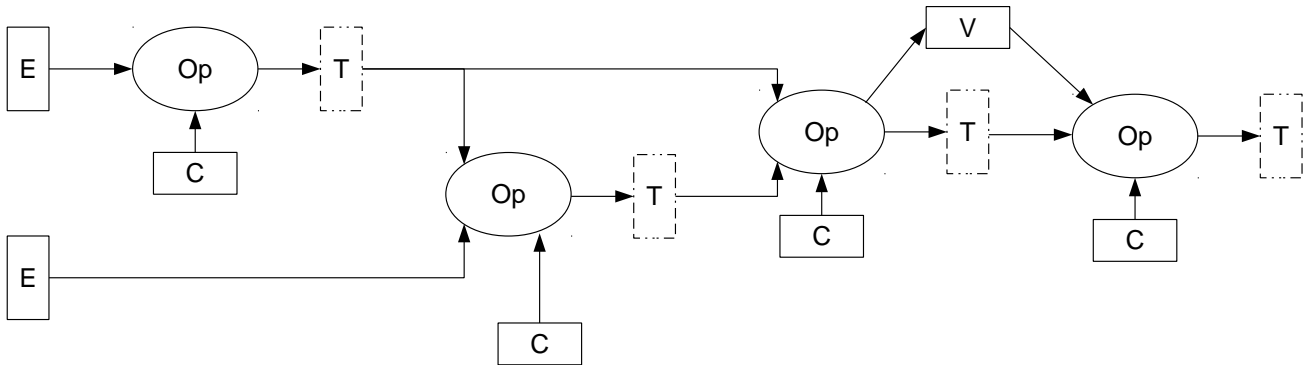


Figure 1. An example computational graph: squares denote tensor data (E: external, C: constant, V: variable, T: regular tensor), ellipses denote operations

Computational graphs describe a *single step* of neural network computation, that is, the computation performed upon a *single input* (or single batch of inputs) is fed to the graph. This trivially describes feedforward networks, but not all networks are that simple. Recurrent networks introduce dependencies among steps by allowing some tensor values computed in one step be used in the next step, therefore letting the result of the overall computation depend on a *sequence of inputs*. In order to maintain a clear and easy-to-validate description, the computation of each step is described by an acyclic graph, and the amount of cyclic dependency required for recurrent networks is achieved via variable tensors that can be updated in each step and retain their values between consecutive steps. To achieve this, a special operation is introduced to update variable



tensors (see [Variable Updates](#)).

## 2.2. Description of Data

A multi-dimensional array can be described by its number of dimensions (rank) and its *extent* in each dimension (its *shape*). Conceptually, a multi-dimensional array is infinite dimensional, the irrelevant (trailing) dimensions being of extent 1 (hereafter referred to as *singleton* dimension). In this description, we do not restrict the number of dimensions, and consider the dimensions that are not explicitly described to be singleton. Of course, in practice, an implementation restricts the number of *supported* dimensions. The minimum number of supported dimensions is 2. Dimensions are indexed starting from 0. The product of all extents is called the *volume* of the tensor (note that trailing singleton dimensions do not effect the volume).

In the computational graph, each tensor must have a well-defined shape. The operations `external`, `constant` and `variable` define the shape of the introduced tensors explicitly, thus providing shape for those tensors that constitute the starting points of the computation. All other operations define the shape of their result tensors as a function of the shape of their input tensors, this way propagating shape information through the computational graph.

In order to describe the structure of a computational graph, no actual data is needed. However, when the graph is to be executed, actual data is required for those tensors that serve as parameters to the computation, such as variables that have a previously computed value. Thus, apart from the structural aspects of data nodes, their actual data also needs to be defined somewhere. The Specification introduces a binary format for describing the contents of individual tensors.



Consumers of an NNEF document are free to replace the shape of external tensors and propagate shapes accordingly as long as it does not lead to invalid arguments of operations in the computational graph.

## 2.3. Description of Operations

Apart from data nodes, an operation node may have attributes that detail the exact computation performed. Operation nodes have well-defined semantics that specify the mapping of input(s) to output(s) including the shape and content of the output tensors depending on those of the input tensors.

Computational graphs may be described in a *flat* or in a *compositional* manner. In a compositional description an operation may be expressed in terms of other, simpler operations. Hence, the description of a computational graph may be organized into a hierarchy. A flat description does not group operations into larger units.

NNEF graph description syntax can be divided into two parts: a core part that is required for a flat description, and an extension part that can describe compound operations. In case of a compositional description, operations fall into two major categories:

¥ *primitive* : operations that cannot be expressed in terms of other operations

¥ *compound* : operations that can be built from other operations

Compound operations can still be considered valid by a flat description, however, they are also treated as atomic primitives.

Below is a graphical illustration of how larger and larger sub-graphs are built, until the whole graph is defined:

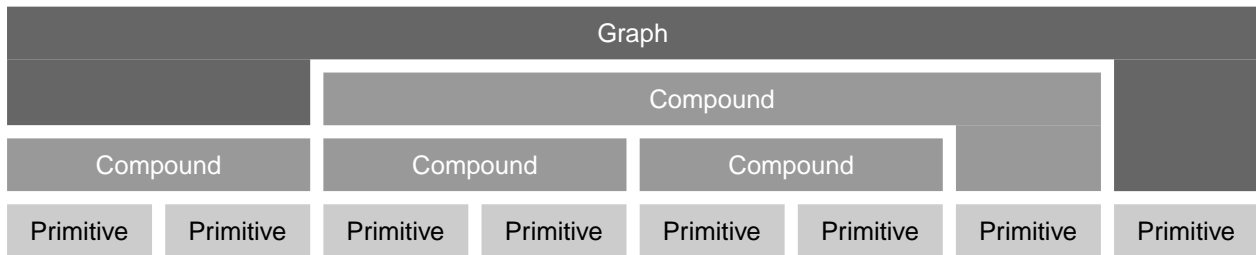


Figure 2. Hierarchical building of compound fragments and the graph

A compositional description is useful as it conveys higher order structure in the graph, grouping frequently occurring sub-graphs. This facilitates a compact description, conveys more information than a flat description, and execution engines may leverage such structural information. However, a compositional description is more complex, more difficult to compile. Thus, where a flat description is sufficient, it is possible to use just the appropriate sub-set of NNEF syntax.

When describing operations, primitives need to specify their input-output mapping, including the computation they perform and the shapes and data-types of their results as a function of their inputs. The semantics of these primitive operations are expressed by mathematical formulae. On the other hand, compound operations are built from primitives, and the way they are built provides them with semantics that can be derived from the primitives. This way, the whole computational graph will have well-defined semantics.

In order to be able to describe compound operations in terms of other operations (primitive or compound), a *procedural* notation is used. Popular deep learning frameworks typically utilize a general-purpose scripting language (such as Python). The present description of graph structure mimics a simple subset of such scripting languages, built around a graph *fragment* construction which lets a parameterized operation be specified in terms of other lower level operations. This methodology can be applied to describe graphs of whole neural networks in a compact manner.

## 2.4. Overview of Graph Description and Usage

The purpose of this format is to describe a computational graph that can ultimately be executed, however, the format itself does not define an execution model, only the structure and the data parameters of the graph. In order to do so, a simple textual format describes the structural aspects of the computational graph. Accompanying this textual format is a data storage format that describes external tensor parameters of the computational graph.

This document is structured as follows. Chapter [Formal Description](#) details the [syntax](#) and the [semantics](#) of the textual format. Chapter [Operations](#) describes a set of primitive and compound operations from which computational graphs can be built, along with their parametrization and semantics. Chapter [Storing Network Data](#) describes the binary format to store network weights.

To give a clearer picture of how the format may be used, we describe how these pieces may be fit

together to compile and execute a computational graph.

### 2.4.1. Graph Compilation and Execution

The compilation of a computational graph starts from:

- ¥ A byte stream (textual) describing the computational graph as a sequence of [operations](#).
- ¥ An optional byte stream (binary) containing the data of [serialized](#) external tensor parameters to the graph.

On a conceptual level, the compilation process may have the following steps:

- ¥ Parse the structural description, check its validity according to the rules in [Formal Description](#).
- ¥ If the description is compositional, expand the hierarchy of operations into primitives, by evaluating compile-time expressions of attributes, resulting in a flattened structure.
- ¥ Propagate tensor shapes, perform argument checking for operations as described in [Operations](#).

An implementation is not restricted to compile a graph of primitive operations as listed in [Operations](#), but instead it may implement operations such as those in the [Compound Operations](#) as atomic ones for improved efficiency. After building an executable graph, an implementation may optimize it for example by removing intermediate buffers and unnecessary operations or merging sequences of operations.

After the compilation process, the graph execution may have the following steps:

- ¥ Load previously serialized data or feed initial values to tensors declared as variables.
- ¥ In each cycle, feed values to tensors declared as external inputs, execute required operations, read out outputs.
- ¥ Save updated variables if necessary.

Note again that the exchange format does not define an execution model of the computational graph being described. The above is just an example of how it might be used.

## 2.5. Glossary of Terms

The following terms are used frequently in the Specification and are listed explicitly here:

attribute

A non-tensor parameter to operations that define further details of the operation. Attributes are of primitive types whose values are known at graph compilation-time, and hence expressions of attributes can be evaluated at graph compilation-time.

compound operation

An operation that is defined in terms of other operations. Its semantics are defined via the composition of the operations that it is defined by.

computational graph

A graph with nodes that are either operations or tensors. Operation nodes are connected to tensor nodes only, and vice versa.

#### custom operation

A primitive operation defined without a body in an NNEF document. It is up to the consumer to know the semantics (mathematical formula and shape information) of such an operation. These are not defined by the specification or the NNEF document itself.

#### graph fragment

A sub-graph in the whole graph (network). A fragment can be described by a set of operations interconnected by tensors.

#### graph compilation-time

The time when the graph is built before execution. Analogous to compilation-time for programming languages.

#### graph execution-time

The time when the graph is run (possibly multiple times) after building. Analogous to run-time for programming languages.

#### operation

A mapping of input tensors to output tensors.

#### primitive operation

An operation that is not defined in terms of other operations. Its semantics are defined via mathematical formulae.

#### rank (of a tensor)

The number dimensions of a tensor explicitly specified in a shape or implicitly defined by shape propagation. Note that a shape explicitly defined as (5,1) has rank 2, even though its last dimension is singular.

#### row-major order

A generalization of row-major data layout of matrices to multi-dimensional arrays. Data is laid out in an order where multi-indexing varies fastest along the last dimension, and slowest along dimension 0. Note that the definition is valid for conceptually infinite dimensional data as well, since the trailing singleton dimensions only introduce trailing 0 indices in the conceptually infinite multi-index.

#### shape (of a tensor)

A list of integers defining the extents of a tensor in each relevant dimension.

#### tensor

A multi-dimensional array of scalars that represents data flow in the graph. The number of dimensions is conceptually infinite; the insignificant trailing dimensions are 1 (singleton dimension). The minimal number of actually supported dimensions by an implementation is 2.

#### variable tensor

A tensor whose value can be updated by a designated operation. All other tensors are conceptually immutable; each operation generates a new tensor.

volume (of a tensor)

An integer value that is the product of extents of a shape.

# Chapter 3. Formal Description

This chapter provides a formal description of the exchange format for the structural aspects of the computational graph. It is a simple notation with the goal to describe the building of computational graph fragments from lower level building blocks, ultimately arriving to the description of whole network graphs.

A grammar in Backus-Naur Form (BNF) is used to describe the [syntax](#) of the notation. First, the [lexical elements](#) of the grammar are defined, and the [constraints](#) associated with valid computational graphs are also enumerated.

## 3.1. Lexical Elements

The description format is made up from the following lexical entities:

### <identifier>

An identifier is an alphanumeric sequence of ASCII characters that may also contain the underscore character. More specifically, identifiers must consist of the following ASCII characters: `_`, `[a-z]`, `[A-Z]`, `[0-9]`. The identifier must not start with a digit.

### <numeric-literal>

A numeric literal consists of an integer part, an optional decimal point (`.`) and a fractional part, an `e` or `E` and an optionally signed integer exponent. The integer, fractional and the exponent parts must each consist of a sequence of decimal (base ten) digits (`[0-9]`). The literal may be preceded by an optional `-` (minus) sign. In case of flat syntax, the minus sign is interpreted as being part of the literal value. In compositional syntax, the minus sign is interpreted as a unary minus operator that precedes the numeric literal, and hence is not part of the literal value. The end result in either case is the same.

### <string-literal>

A string literal is a sequence of characters enclosed within `'` or `"` characters. The end and start quotes must match. Any printable ASCII character may appear within the string, except for the start quote character, which must be escaped by the `\` character. The `\` character must also be escaped with the `\` character.

### <logical-literal>

Logical literals are the values `true` and `false`.

### <keyword>

The following alphabetic character sequences have special meaning with respect to the description syntax and thus must not be used as identifiers: `version`, `extension`, `graph`, `fragment`, `tensor`, `integer`, `scalar`, `logical`, `string`, `shape_of`, `length_of`, `range_of`, `for`, `in`, `yield`, `if`, `else`.

### <operator>

The following character sequences have special meaning as operators in mathematical expressions: `+`, `-`, `*`, `/`, `^`, `<`, `<=`, `>`, `>=`, `=`, `!=`, `&&`, `||`, `!`.

### Syntactic characters

The following characters have special syntactic meaning: `(`, `)`, `[`, `]`, `{`, `}`, `:`, `=`, `,`, `;`, `->`.

## White spaces

White space characters may be inserted between any lexical entities. They include the space character, the control characters representing horizontal tab, vertical tab, form feed and new-line.

## Comments

Comments are introduced by the `#` symbol, and last until the end of the line (either until form feed or new line characters).

## 3.2. Syntax

The central concept in computational graphs is an operation that maps input tensors into output tensors. Operations need to be declared, and a computational graph description is built from the declared operations. Therefore, we separate the description into two key parts:

- ¥ The declaration of possible operations for building graphs. Operations have a name and a list of parameters and results. Formal parameters are [typed](#), identifying what kind of expressions can be substituted in their place.
- ¥ The actual graph description consists of a list of operation invocations that are validated against the declarations. The operations are invoked by referencing their names and supplying arguments in place of their formal parameters.

Furthermore, the description of syntax is separated into the constructions sufficient for a *flat* description and extensions required for *compositional* descriptions.

The following BNF notation introduces the description syntax in a more formal manner. Everything defined by `::=` below is part of the BNF description, and constitutes valid syntax. Anything outside of the grammar defined by these BNF rules is considered invalid syntax.

### 3.2.1. Graph Definition

A graph definition consists of a graph declaration and its body. The graph declaration has a list of parameters and results.

```
<graph-definition> ::= <graph-declaration> <body>
<graph-declaration> ::= "graph" <identifier> "(" <identifier-list> ")"
                        "->" "(" <identifier-list> ")"
<identifier-list> ::= <identifier> ("," <identifier>)*
```

The graph definition itself consists of a list of assignments, where the left-hand-side of the assignment is an identifier expression (single identifier, tuple or array), and the right-hand-side is an operation invocation.

```
<body> ::= "{" <assignment>+ "}"
<assignment> ::= <value-expr> "=" <invocation> ";"
```

An invocation consists of an identifier and a list of arguments:

```
<invocation> ::= <identifier> ["<" <type-name> ">"] "(" <argument-list> ")"  
<argument-list> ::= <argument> ("," <argument>)*  
<argument> ::= <rvalue-expr> | <identifier> "=" <rvalue-expr>
```

Expressions may be literals, identifiers, arrays and tuples. It is necessary to differentiate between left-value and right-value expressions.

Left-value expressions are allowed on the left-hand-side of assignments:

```
<array-lvalue-expr> ::= "[" [<lvalue-expr> ("," <lvalue-expr>)* ] "]"  
<tuple-lvalue-expr> ::= "(" <lvalue-expr> ("," <lvalue-expr>)+ ")" |  
Ê <lvalue-expr> ("," <lvalue-expr>)+  
<lvalue-expr> ::= <identifier> | <array-lvalue-expr> | <tuple-lvalue-expr>
```

Right-value expressions are allowed on the right-hand-side of assignments (as argument values):

```
<array-rvalue-expr> ::= "[" [<rvalue-expr> ("," <rvalue-expr>)* ] "]"  
<tuple-rvalue-expr> ::= "(" <rvalue-expr> ("," <rvalue-expr>)+ ")"  
<rvalue-expr> ::= <identifier> | <literal> | <array-rvalue-expr> | <tuple-rvalue-expr>  
  
<literal> ::= <numeric-literal> | <string-literal> | <logical-literal>
```

Invocations may have multiple results (if the operation defines multiple results). In this case, the returned expression is a tuple, and the left-hand-side expression must also be a tuple.

As an example, using the declarations above, we may define part of a graph as:

```
graph barfoo( input ) -> ( output )  
{  
Ê input = external( shape = [1,10] );  
Ê intermediate, extra = bar( input, alpha = 2 );  
Ê output = foo( intermediate, size = [3,5] );  
}
```

In the above example, `external` is an operation used to introduce tensors that receive their data from an external source (see [Tensor Introducing Operations](#)), and exemplary operations `bar` and `foo` are defined below.

### 3.2.2. Fragment Definition

The following syntax elements must be enabled by the extension `KHR_enable_fragment_definitions`.

A fragment is similar to the graph in that its body is defined by a list of assignments, but its declaration allows typed formal parameters and results.



```
<fragment-definition> ::= <fragment-declaration> (<body> | ";" )
```

An fragment declaration is introduced by the **fragment** keyword, has a name, a parameter list and a result list. Parameters are explicitly typed and may have default values. Results are introduced after the **->** symbol.

```
<fragment-declaration> ::= "fragment" <identifier> [<generic-declaration>]
    "(" <parameter-list> ")" "->" "(" <result-list> ")"
<generic-declaration> ::= "<" "?" ["=" <type-name>] ">"
<parameter-list> ::= <parameter> ("," <parameter>)*
<parameter> ::= <identifier> ":" <type-spec> ["=" <literal-expr>]
<result-list> ::= <result> ("," <result>)*
<result> ::= <identifier> ":" <type-spec>
```

Default values are literal expressions built from literals only:

```
<array-literal-expr> ::= "[" [<literal-expr> ("," <literal-expr>)* ] "]"
<tuple-literal-expr> ::= "(" <literal-expr> ("," <literal-expr>)+ ")"
<literal-expr> ::= <literal> | <array-literal-expr> | <tuple-literal-expr>
```

A type specification may denote a primitive type, an array type or a tuple type.

```
<type-name> ::= "integer" | "scalar" | "logical" | "string" | "?"
<tensor-type-spec> ::= "tensor" "<" [<type-name>] ">"
<array-type-spec> ::= <type-spec> "[" "]"
<tuple-type-spec> ::= "(" <type-spec> ("," <type-spec>)+ ")"
<type-spec> ::= <type-name> | <tensor-type-spec> |
    <array-type-spec> | <tuple-type-spec>
```

For example, the following lines show some operation declarations:

```
fragment foo( input: tensor<scalar>, size: integer[] = [1] )
-> ( output: tensor<scalar> )

fragment bar( input: tensor<scalar>, alpha: scalar = 0.5 )
-> ( output: tensor<scalar>, extra: tensor<scalar>[] )
```

The result types of fragments must be all tensors.

### 3.2.3. Operator Expressions

The following syntax elements must be enabled by the extension **KHR\_enable\_operator\_expressions**.

The syntax enables more complex expressions to be used as right-value expressions. These expressions allow for compile-time arithmetic and argument composition.

Various arithmetic, comparison and logical operators can be used to build binary expressions:

```
<comparison-operator> ::= "<" | "<=" | ">" | ">=" | "==" | "!=" | "in"
<binary-arithmetic-operator> ::= "+" | "-" | "*" | "/" | "^"
<binary-logical-operator> ::= "&&" | "||"
<binary-operator> ::= <comparison-operator>
Ê                               | <binary-arithmetic-operator>
Ê                               | <binary-logical-operator>
```

A handful of unary operators are also available for arithmetic and logical expressions:

```
<unary-arithmetic-operator> ::= "+" | "-"
<unary-logical-operator> ::= "!"
<unary-operator> ::= <unary-arithmetic-operator>
Ê                   | <unary-logical-operator>
```

Operator expressions can then be built using unary and binary operators and parenthesizing:

```
<unary-expr> ::= <unary-operator> <rvalue-expr>
<binary-expr> ::= <rvalue-expr> <binary-operator> <rvalue-expr>
<paren-expr> ::= "(" <rvalue-expr> ")"
```

The if-else expression implements branching by selecting one of two expressions depending on a condition.

```
<if-else-expr> ::= <rvalue-expr> "if" <rvalue-expr> "else" <rvalue-expr>
```

The array comprehension expression implements a form of looping. It generates an array by iterating one or more others, optionally filtering the resulting items.

```
<loop-iter> ::= <identifier> "in" <rvalue-expr>
<loop-iter-list> ::= <loop-iter> ("," <loop-iter>)*
<comprehension-expr> ::= "[" "for" <loop-iter-list> ["if" <rvalue-expr>]
Ê                               "yield" <rvalue-expr> "]"
```



When a comprehension expression contains an **if** condition, the **if** is interpreted as part of the comprehension expression and not as part of an if-else expression inside the comprehension expression following the **in** keyword.

Subscripting expressions can reference a single entry in an array, or a range of entries, in which case the start (inclusive) and end (exclusive) of the range is separated by **:**. Both the start and the end are optional, in which case 0 or the length of the array is taken, respectively.

```
<subscript-expr> ::= <rvalue-expr> "[" (<rvalue-expr> |
Ê                                     [<rvalue-expr>] ":" [<rvalue-expr>]) "]"
```

A few special keywords can be used for built-in functions.

```
<builtin-name> ::= "shape_of" | "length_of" | "range_of"
Ê               | "integer" | "scalar" | "logical" | "string"
<builtin-expr> ::= <builtin-name> "(" <rvalue-expr> ")"
```

Finally, extended right-value expressions are the union of all the above constructions (including the ones in the definition of basic right-value expressions and invocations). Assignments may contain any right-value expression, not only invocations:

```
<rvalue-expr> ::= <identifier>
Ê               | <literal>
Ê               | <binary-expr>
Ê               | <unary-expr>
Ê               | <paren-expr>
Ê               | <array-rvalue-expr>
Ê               | <tuple-rvalue-expr>
Ê               | <subscript-expr>
Ê               | <if-else-expr>
Ê               | <comprehension-expr>
Ê               | <builtin-expr>
Ê               | <invocation>

<assignment> ::= <lvalue-expr> "=" <rvalue-expr> ";"
```

### 3.2.4. The Whole Document

The NNEF structure description consists of a version info, an optional list of extensions used, an optional list of operation definitions and a top-level graph definition. An NNEF document that does not contain operation definitions and extended expressions is said to be *flat*. A graph definition must be always present.

```
<document> ::= <version> <extension>* <fragment-definition>* <graph-definition>
```

The version info is introduced by the **version** keyword, and is defined by a real-number numeric literal as major and minor versions separated by a dot:

```
<version> ::= "version" <numeric-literal> ";"
```

Extensions can be specified after the **extension** keyword:

```
<extension> ::= "extension" <identifier>+ ";"
```

Three types of extensions are distinguished, and this is reflected in naming conventions:

!

¥ Khronos extensions use the format: `KHR_extension_name`.

¥ Cross-vendor extensions use the format `EXT_extension_name`.

¥ Vendor-specific extensions use the format `VENDOR_extension_name`, where VENDOR is the actual name of the vendor.

Here is a short example for illustration:

```
version 1.0;
extension KHR_enable_fragment_definitions;

fragment foo( input: tensor<scalar>, flag: logical ) -> ( output: tensor<scalar> )
{
    output = ...
}

fragment bar( input: tensor<scalar>, param: scalar ) -> ( output: tensor<scalar> )
{
    output = ...
}

graph foobar( input ) -> ( output )
{
    input = external( shape = [4,10] );
    hidden = foo( input, flag = true );
    output = bar( hidden, param = 3.14 );
}
```

Note, that it is possible to build networks solely from predefined operations (see [Operations](#)), which need not be defined in an actual document description. Therefore, fragment definitions are usually unnecessary. Furthermore, the graph definition body can usually be written without using operator expressions; they are most useful for defining compound operations. Hence, networks without custom operations can usually be written using flat syntax only.

### 3.3. Semantics

The following subsections define when a [syntactically](#) valid document made up of fragment definitions is also semantically well-defined. Semantic validity rules describe the proper definition and invocation of fragments, including proper naming and referencing, argument number, argument types, argument ranges, proper usage of declared parameters and local identifiers.

### 3.3.1. Type System and Type Checking

In the grammar defined in [Syntax](#), formal parameters of operations are explicitly typed. Furthermore, literal constants also have an implicitly defined type. The types of expressions are derived from the types of their arguments.

Types can either be primitive or compound. Compound types are composed from primitives or from other compound types.

#### Primitive Types

The following primitive-types are defined:

- ¥ `integer` : a signed integer value used to describe tensor shapes, dimensions and indices into tensors
- ¥ `scalar` : a real value used to describe data and parameters in operations
- ¥ `logical` : a logical value typically used to describe flags and branching conditions
- ¥ `string` : a general character sequence, used to denote enumerations and names

The primitive types are used to denote parameters whose values are known in compile-time.

#### Compound Types

Compound types follow a few construction templates:

- ¥ *Tensor types* are built from a single data-type, which must be a primitive type. For example `tensor<scalar>` is a tensor of scalars. Tensors represent data on which run-time computations is performed. A tensor type may be *unbound*, that is, without data-type, using the `tensor<>` syntax.
- ¥ *Array types* are built from a single item type. Each item in the array has the same type. For example, `integer[]` is an array of integers.
- ¥ *Tuple types* are built from multiple item types and always have a fixed number of items. For example, `(integer, scalar)` is a tuple of an integer and a scalar.

A type is said to be an *attribute* type if it does not contain tensor types. Tuples must not contain both tensor and non-tensor item types at the same time to separate run-time data parameters and compile-time attribute values.

The item type of a compound type may also be a compound type (except for tensors), thus enabling arrays of arrays, arrays of tuples or a tuple that contains an array. For example `scalar[][]` is a 2-dimensional scalar array (where each sub-array may be of different length), `(integer, scalar)[]` is an array of tuples, and `(integer[], scalar)` is a tuple that contains an array and a scalar.

Unbound tensors cannot be the result of an operation, they can only be used to declare inputs whose data-types do not need to be checked or agreed with other tensors.



It is up to the implementation of inference engines how tensor data types are represented.

## Generic Types

The syntax offers a limited way of constructing generic types via the use of `?` symbol in place of a primitive type. It may also be used in compound types to denote for example generic tensors as `tensor<?>` or generic arrays as `?[]`. The generic symbol can be used in fragment declarations to declare that an operation can be applied to tensors of any data-type. For example

```
fragment buzz<?>( input: tensor<?>, param: ? ) -> ( output: tensor<?> )
```

declares an operation `buzz` to be applicable to tensor of any data-type. The definition also implies that the data-type of the parameter `input` must be the same as the type of parameter `param`, and that the data-type of result `output` will be inherited from input.

When invoked, the generic data-type `?` may be inferred from the actual arguments. However, there may be fragments that do not have inputs of generic type, only the output is generic. For this case, the generic type may have an optional default value.

```
fragment fizz<? = scalar>( shape: integer[] ) -> ( output: tensor<?> )
```

In this case, the generic parameter must be explicitly supplied, or if not, the default value is used.

```
a = fizz<integer>(shape = [...])    # explicit integer data-type
b = fizz(shape = [...])            # default scalar data-type
```

In case the generic parameter needs to be propagated in a compound operation, it can be done using `<?>` after the name of the operation:

```
fragment fizz<? = scalar>( shape: integer[] ) -> ( output: tensor<?> )
{
    output = buzz<?>(...);
}
```

## Types of Literal Constants

The types of literal constants are as follows:

- ¥ The type of `<numeric-literal>` is either `integer` or `scalar` depending on whether it denotes an integer or a real value, respectively.
- ¥ The type of `<string-literal>` is `string`.
- ¥ The type of the constants `true` and `false` is `logical`.
- ¥ The type of the literal `[]` is the special type *empty-array-type*. This type cannot be declared explicitly.

## Type Casting

Only the following implicit type casts are allowed (all others are disallowed):

- ¥ Primitive types can be cast to the tensor types of the corresponding data-type.
- ¥ Any primitive can be cast to the generic type `?`, and any (bound) tensor type can be cast to the generic type `tensor<?>`.
- ¥ Any tensor type can be cast to the unbound tensor type `tensor<>`.
- ¥ An array type can be cast to another array type if its item type can be cast. The empty-array-type can be cast to any other array type, but no other array type can be cast to the empty-array-type.
- ¥ A tuple type can be cast to another tuple type if they have the same number of items and the corresponding item types can be cast to those of the other tuple type.

Two types have a common type if either of the two can be cast to the other one.

When parameters of primitive type are substituted in place of tensor parameters, they behave as constant tensors of singleton shape.

When an expression is substituted in place of a formal parameter in an invocation of an operation or is assigned to the result of a fragment, the expression type must be equal or castable to the formal parameter or result type; otherwise the invocation is invalid.

Furthermore, explicit type casting between primitive types can be forced using [built-in functions](#) where necessary.

### 3.3.2. Graph and Fragment Definition

#### Declarations

Each fragment declaration must have a unique name. Fragments are identified only based on their name. A valid declaration must satisfy the following criteria:

- ¥ Formal parameter names and result names must be unique within a declaration, that is, valid declarations must not contain multiple formal parameters or results with the same name (of course, different declarations may use the same formal parameter names).
- ¥ Parameters of tensor type must precede attributes.
- ¥ Fragment results must be all tensors.
- ¥ The default value expression type must match the type of the formal parameter.

Generic fragments may be defined by appending `<?>` after the name of the fragment. The symbol `?` is used to refer to the generic type parameter within the fragment declaration and body. The generic type parameter may have an optional default value using `<? = ...>` syntax. See [Generic Types](#) for further details.

If a fragment has at least one generic parameter or result type, it must be declared as generic. In return, if a fragment is declared as generic, it must have at least one generic parameter or result type.

## Invocations

Invocations must assign a unique argument value to each formal parameter that does not have a default value. Formal parameters with default values can also be assigned a different value.

There are two ways to assign values to formal parameters: position based (*positional argument*) and name based (*named argument*). Positional arguments correspond to formal parameters in the order of their declaration. Named arguments may occur in any order, independent of the order of declaration of formal parameters. Only tensor arguments may be positional, attribute values must be named arguments (see [Type System and Type Checking](#)).



It is deprecated to use named arguments for tensor parameters in an invocation. To clearly separate tensor arguments and non-tensor attributes, tensors should be positional arguments, and attributes should be named arguments.

A valid invocation must satisfy the following criteria:

- ¥ The operation name in the invocation must correspond to a declared operation.
- ¥ An invocation must not have more arguments than the number of formal parameters in the declaration.
- ¥ Each formal parameter that does not have a default value must be assigned an argument value.
- ¥ Positional arguments must precede named arguments.
- ¥ Each named argument must have a name that corresponds to a formal parameter declared for the operation.
- ¥ Named arguments must be unique, that is, each parameter name may occur only once in an invocation.
- ¥ A named argument must not refer to a formal parameter that is also assigned by a positional argument.
- ¥ Generic fragments may be called with an explicit type specifier using `name<type>` syntax. Furthermore, `name<?>` syntax may be used when a generic parameter type needs to be propagated inside the body of a generic fragment. If a generic fragment has a default value for its generic type and is invoked without an explicit generic argument, the default value of the generic type is substituted.

In an invocation, formal parameters that have a default value and are not assigned any value get their default values substituted.

If a fragment is generic and an explicit generic parameter type is not specified, and has no default value for its generic type, then the value of the generic parameter is deduced from the invocation arguments. The deduction must result in a unique value. If it results in multiple or no candidates, the invocation is invalid.

## Assignments

The right-hand-side of an assignment can be any kind of expression, however, the left-hand-side must be an identifier or an array or tuple (or any such combination) of identifiers to which the results are unpacked. That is, an array in a tuple is permitted, but invocations and constant



expressions are not allowed on the left-hand-side.

In case the right-hand-side is an invocation, the left-hand-side expression must be structurally equivalent to the result type of the invoked operation. For example, if the operation results in an array of tensors, the left-hand-side must be an array of identifiers. This is required so that the length of the resulting array is known from the assignment without knowing the exact semantics of the operation.

## Identifier Usage in the Graph and Fragment Body

The rules for valid formal identifier usage are as follows:

- ¥ Parameters of a *fragment* must not appear on the left-hand side of an assignment within the fragment body, they can only be used as arguments of expressions on the right-hand side of assignments.
- ¥ Parameters of a *graph* (which are implicitly of type tensor) must be defined as the result of an **external** operation. The **external** operation must not be used in a fragment, and all tensors that are results of **external** operations must be defined as graph parameters (one-to-one mapping between graph parameters and externally defined tensors).
- ¥ Results of a fragment must be assigned exactly once within the fragment body.
- ¥ Local identifiers within a fragment or within the main graph must first appear on the left-hand side of an assignment before used as arguments in expressions on the right-hand side of subsequent assignments.
- ¥ The same identifier must not be used on the left-hand side of assignments more than once (within a fragment, or within the main graph).
- ¥ All individual identifiers in the *graph* body must be of type tensor. That is, identifiers of type tensor array, tensor tuple and primitive types are not allowed. When tensor arrays or tuples are used on the left-hand-side of assignments, they must be explicitly constructed from individual tensor identifiers via array or tuple expressions.
- ¥ The **variable** and **update** operations must not be used inside a fragment, only inside the graph body (the results of a compound operation should only be defined by its inputs, and it should not have any side effects).

The above rules ensure that the resulting graph is acyclic and the order in which the operations are written in the body results in a valid topological ordering of the graph. However, it is not the only valid ordering, and operations can be reordered or executed in parallel as long as the above constraints are satisfied.

The purpose of disallowing identifiers of type tensor array or tensor tuple in the graph body, is to let each tensor be identifiable by an individual name.

## Argument Validity

Each operation may further restrict the set of allowed parameters beyond what is generally considered valid according to **type checking**. For example, operations may restrict their attributes to be in specific ranges. Operations that have multiple tensor parameters may define tensor shape agreement rules, and they also define the shapes of their results. These rules are specific to each

operation, and are described separately for each primitive in [Operations](#).

### 3.3.3. Building Expressions

The expressions described here mainly use extended syntax (except for simple array and tuple construction). The purpose of such expressions is two-fold:

- ¥ One is to serve as a short-hand notation for certain operations. For example, for tensors **a** and **b**, **a + b** is equivalent to **add(a, b)**.
- ¥ The other is to build or calculate attribute values of operations. For example, an operation may require an array of integers, and that can be built with an expression such as **[1, 2, 3]**.

Thus, expressions are either of frequently used arithmetic, comparison or logical operators, or serve the purpose of manipulating a data structure, such as an array or a tuple to serve as arguments for operations. When manipulating data structures, the typical set of operations that are required are building a data structure from its components or parts, and dissecting a data structure to its components or parts.

The following subsections describe built-in operators and data-structure manipulator notations.

#### Built-in Operators

Arithmetic, comparison and logical expressions can be built from identifiers and constants (see [Syntax](#)). An expression can be a constant expression, which is built only from primitive types, and it can be non-constant expression, in which case it can be mapped to primitive operations (see [Operations](#)). The following table summarizes the allowed argument types and the resulting type for each operator. All operators are also applicable to tensors, hence they are not listed here explicitly. In this case, the result is also a tensor (of the appropriate data-type). If at least one of the arguments is a tensor, the operation is mapped to the appropriate fragment. In this case, argument types must be checked in the same way as if the fragment was explicitly called.

Arithmetic operators that are applicable to **scalar** are also applicable to **integer** arguments (but not mixed), in which case the result is also an **integer**. Comparison operators that are applicable to any primitive type (marked by **?** below just like generic types) must have arguments of the same type.

*Table 1. List of built-in operators; parameter and result types, and mapping to fragments*

Operator	Argument types	Result type	Fragment
+	scalar	scalar	copy
-	scalar	scalar	neg
+	scalar, scalar	scalar	add
-	scalar, scalar	scalar	sub
*	scalar, scalar	scalar	mul
/	scalar, scalar	scalar	div
^	scalar, scalar	scalar	pow
<	scalar, scalar	logical	lt

Operator	Argument types	Result type	Fragment
<=	scalar, scalar	logical	le
>	scalar, scalar	logical	gt
>=	scalar, scalar	logical	ge
==	?, ?	logical	eq
!=	?, ?	logical	ne
!	logical	logical	not
&&	logical, logical	logical	and
	logical, logical	logical	or

Uses of the above operators with argument types other than those listed above result in invalid expressions.

The semantics of arithmetic operators `+`, `-`, `*`, `/`, comparison operators `<`, `<=`, `>`, `>=`, `==`, `!=` and logical operators `&&`, `||`, `!` is as per their common definitions. Furthermore, the binary operator `in` tests if an item is contained in an [array](#).

The precedence of the operators is as follows, from lowest to highest, equal ones grouped in curly braces: `{ in }`, `{ &&, || }`, `{ <, <=, >, >=, ==, != }`, `{ +, - }`, `{ *, / }`, `{ ^ }`.

## Arrays

When building arrays, items must be of the same type or it must be possible to [cast](#) all items to a common type.

The simplest way to build an array is to enumerate its items, such as `[a, b, c]`. Alternatively, arrays can be built by concatenating two arrays using the `+` operator, such as `[a, b] + [c, d]` resulting in `[a, b, c, d]`, or `x + y` where `x` and `y` are themselves arrays. As a generalization of concatenation, the `*` operator can be used to duplicate an array several times. For example, `[a, b] * 2` results in `[a, b, a, b]`.

To access an item or range of items in an array, the subscript operator `[]` can be used. Array indexing starts from 0, and goes until the length of the array minus 1. There are two types of subscript expressions, depending on the expression inside the `[]`:

- ¥ If the subscript expression is a single expression of type `integer`, the result is a single item of the array and the type of the subscript expression is that of the item type of the array. For example if `a = [1, 2, 3]` then `a[1]` equals 2.
- ¥ If the subscript expression is a range (two expressions of type `integer` delimited by `:`), then the result is a sub-sequence of the array and the type of the subscript expression is the same as the array type. The start of the range is inclusive; the end is exclusive. The range may be open at both ends (by omitting the expressions before or after the `:`). If it is open at the beginning, it is defined to start from 0. If it is open at the end, the end is defined to be the length of the array. If the beginning is greater than or equal to the end, the result is the empty array. For example let `a = [1, 2, 3]`, then `a[0:2]` is equivalent to `a[:2]` and equals `[1, 2]`, or `a[1:3]` is equivalent to `a[1:]`

and equals `[2, 3]`. Furthermore, `a[2:2]` equals `[]`.

The `in` operator can be used to test whether an item is contained in an array, returning a logical value. For example: `a = b if i in [2,4,6] else c`. The values are matched using deep comparison, i.e. arrays and tuples match if their items match.

## Tuples

Tuples can be constructed from any expression by enumerating the items separated by the `,` operator. For example, `a,b` is a tuple, or optionally, for better legibility, a tuple can be parenthesized, as in `(a,b)`.

If an expression is of tuple type, it can be unpacked by assigning it to a tuple containing identifiers to which the tuple items are assigned. For example, if `t` is a tuple of three items, then `a, b, c = t` unpacks the items of the tuple to the identifiers `a`, `b` and `c`. The tuple on the left must have the same number of items as the one on the right. Fragments that have multiple results essentially return a tuple.

Tuple items can also be accessed by subscripting with limitations: subscripts must be integer literals, that is, ranges and index variables or expressions are forbidden. `a, b, c = t` is equivalent to `a = t[0]; b = t[1]; c = t[2]`.

## Strings

Although `string` is a primitive type, strings can be thought of as arrays of characters and some operations can be defined the same way as for arrays. Strings cannot be built from characters as an array expression, however:

- ¥ Strings can be concatenated using the `+` operator and duplicated with the `*` operator.
- ¥ Sub-strings can be created with range indexing. When the subscript expression is a single index, it also results in a string of a single character.

## Built-in Functions

Some built-in functions are used to query information from tensors, arrays and strings. To describe the interface of such functions, the same notation is used for convenience as for declarations. However, these functions must be called with a single positional argument. The `?` in place of the type specifier means that any type may be substituted (note that in these cases, it really means any type, not just any primitive type).

- ¥ `shape_of( x: tensor<> ) -> ( shape: integer[] )` returns the shape of a tensor `x`. The length of the resulting array is the rank of the tensor. In case a non-tensor value is substituted in place of tensor `x`, the empty array corresponding to the singleton shape of rank 0 is returned (deprecated, see below).
- ¥ `length_of( x: ?[] ) -> ( length: integer )` returns the length of the array `x`.
- ¥ `length_of( x: string ) -> ( length: integer )` returns the length of the string `x`.
- ¥ `range_of( x: ?[] ) -> ( range: integer[] )` returns the range from 0 (inclusive) to the length of array `x` (exclusive).

¥ `range_of( x: string ) -> ( range: integer[] )` returns the range from 0 (inclusive) to the length of string `x` (exclusive).

Furthermore, explicit type casting can be forced using the type names of primitive types as unary functions (`scalar`, `integer`, `logical`, `string`):

¥ `scalar( x: logical ) -> ( y: scalar )` returns `1.0` if the passed value is `true` and `0.0` otherwise.

¥ `scalar( x: string ) -> ( y: scalar )` returns the scalar representation of a string if it describes a valid scalar literal value (according to the syntax), otherwise the invocation is invalid.

¥ `scalar( x: integer ) -> ( y: scalar )` returns the same value as passed in, only the type is changed.

¥ `integer( x: logical ) -> ( y: integer )` returns `1` if the passed value is `true` and `0` otherwise.

¥ `integer( x: string ) -> ( y: integer )` returns the integer representation of a string if it describes a valid integer literal value (according to the syntax), otherwise the invocation is invalid.

¥ `integer( x: scalar ) -> ( y: integer )` returns the passed value truncated to the closest smaller integer value.

¥ `logical( x: integer ) -> ( y: logical )` returns `false` if the passed value is `0` and `true` otherwise.

¥ `logical( x: scalar ) -> ( y: logical )` returns `false` if the passed value is `0.0` and `true` otherwise.

¥ `logical( x: string ) -> ( y: logical )` returns `false` if the passed string is the empty string ( `''` ) and `true` otherwise.

¥ `string( x: ? ) -> ( y: string )` returns the string representation of any value of primitive type according to its literal representation in the syntax.

!

The `shape_of` function is deprecated and is discouraged from use. The reason is that it provides syntactic means to access a property of tensors that is not defined via the syntax itself. Furthermore, its definition is problematic in cases where the shape of a tensor is not known in graph compilation time. These result in problems with custom operations and operations with results of dynamic shape for a consumer of an NNEF document. By removing support for the `shape_of` function from NNEF syntax, it becomes possible to de-couple parsing from shape propagation in a consumer of an NNEF document.

## Compile-time Branching

Compile-time branching is achieved via the syntax `z = x if condition else y`. The `condition` must be an expression of type `logical` (thus its value is known at compile time). Furthermore, the evaluation of `x` or `y` should be lazy, that is, after the `condition` is evaluated, only the appropriate one of `x` and `y` should be evaluated, this way allowing the unevaluated to be invalid as well (for example indexing an array out of bounds), which is necessary when expressing certain constructions such as recursion, as in the `add_n` operation (defined in [Compound Operations](#)):

```
fragment add_n( x: tensor<scalar>[] ) -> ( y: tensor<scalar> )
{
  y = x[0] + add_n(x[1:]) if length_of(x) > 0 else 0.0;
}
```

In the above example of recursive addition, when `length_of(x) == 0` both expressions `x[0]` and `x[1:]` would be invalid, but are not considered because of lazy evaluation.

### Compile-time Looping

An obvious way of looping is to use recursion, as in the above example. However, it is often very cumbersome to write and hard to understand.

A simpler way to achieve parallel looping is *array comprehension*, which generates an array by iterating another one and transforming its items. In general, the expression `b = [for i in a yield f(i)]` iterates the array `a` and generates items by applying `f` to each item `i`. The length of the resulting array is equal to that of the iterated array. Multiple loop iterators may exist, such as in `c = [for i in a, j in b yield i + j]`. In this case, the iterated arrays must have the same length. The loop iterator `i` may also be an index running through a range, as in `b = [for i in range_of(a) yield f(i, a[i])]`. Optionally, a condition can be provided to filter the resulting items: `b = [for i in a if c(i) yield f(i)]` outputs an item only if condition `c(i)` evaluates to `true`. In this case, the length of the output is equal to the number of items for which the condition is met.

### Invocation Chaining

Operation invocations can be chained just like function calls in programming languages, in which case the result of an operation is the input to another operation. As an example, the chain `z = g(f(x), y)` is equivalent to the following sequence of invocations

```
t = f(x);
z = g(t, y);
```

However, chaining is only allowed if the operation `f` returns a single tensor, since the left-hand-side of the invocation (the implicit intermediate identifier `t`) must be structurally equivalent to the result type of operation `f`, and that is only satisfied if it returns a single tensor.

Note, that expressions containing multiple operators are also implicitly chained. For example `a = b + c * d` is equivalent to `a = add(b, mul(c, d))`.

### 3.3.4. Exported Identifiers

Certain tensors need to be referenced from outside the main description to be able to attach further information to the graph (such as parameter data or quantization information). There are two types of tensors that are possible to reference: variables and activation tensors.

Variables are declared with an explicit string label (see [Variables](#)), and this label must be globally unique (except for shared weights), hence it can be used for referencing a variable. This label is

used for example to attach tensor data to variables that are serialized.

Activation tensors in the graph body also have a globally unique identifier (the left-hand side of assignments must have previously unused names), hence these can also be used to reference activation tensors, for example to attach quantization information to activations.

Note, that variables can also be part of the main graph description, and hence they may be referenced by two mechanisms (the string label of the variable definition, and the identifier name used in the assignment).

```
variable42 = variable(label = 'block1/conv1/weights', ...);
```

In the above example, the names 'variable42' and 'block1/conv1/weights' refer to the same tensor, albeit for different purposes.

# Chapter 4. Operations

This chapter describes the (primitive) operations that can be used to build neural network computations. The description of operations contains

- ¥ The declaration of each primitive using the syntax introduced in chapter [Formal Description](#).
- ¥ The description of parameters in the declaration.
- ¥ Argument validity constraints related to each primitive, including input-output shape relations.
- ¥ The semantics of the operation, that is, mathematical formulae describing how the outputs are calculated from the inputs and the attributes of the operation.

!

Some operations are described as compounds via other primitives using NNEF syntax. Such a description is purely functional, that is, to describe what the operation must compute. It bears no restriction to its actual implementation in a consumer of an NNEF document; it may be treated as an atomic operation by a consumer, or it may be decomposed to primitives in a different but functionally equivalent way.

Operations can be grouped into a few larger categories, while some operations are one-of-a-kind. The larger groups are:

- ¥ [Tensor Introducing Operations](#)
- ¥ [Element-wise Operations](#)
- ¥ [Sliding-Window Operations](#)
- ¥ [Reduce Operations](#)
- ¥ [Tensor Shape Operations](#)
- ¥ [Region-of-Interest Operations](#)

Some operations treat the first two dimensions in a special way. The first dimension (index 0) is considered the *batch* dimension, the second (index 1) the *channel* dimension. The rest of the dimensions are called the *spatial* dimensions.

## 4.1. Tensor Introducing Operations

The following operations introduce tensors that are not the result of a calculation, such as external inputs to the computation or parameters like weights and biases.

### 4.1.1. External Data Sources

An external data source is a tensor which must be fed to the graph from the outside.



```
fragment external<? = scalar>(
  Ê shape: integer[] )      # the shape of the tensor
-> ( output: tensor<?> )
```

#### *Argument validity*

¥ Items in **shape** must be strictly positive.

#### *Result semantics*

¥ The shape of **output** is equal to **shape**. The rank of **output** is equal to the length of **shape**.

The content of **output** is not defined by the operation. The tensor must be fed with input data before each execution of the graph. The content of the tensor is *not* expected to persist between subsequent invocations of the computational graph.

### 4.1.2. Constants

A constant is a tensor that has a fixed value.

```
fragment constant<? = scalar>(
  Ê shape: integer[],      # the shape of the tensor
  Ê value: ?[] )          # the values to fill the tensor with
-> ( output: tensor<?> )
```

#### *Argument validity*

¥ Items in **shape** must be strictly positive.

¥ The length of **value** must equal the volume implied by **shape** or must be 1.

#### *Result semantics*

¥ The shape of **output** is equal to **shape**. The rank of **output** is equal to the length of **shape**.

¥ **output** is filled with values such that its row-major ordering equals the items in **value**.

Note, that a constant tensor of singular dimensions can be simply written as a numeric literal directly into expressions, so **y = x + constant(shape = [1], value = [3.14])** is equivalent to **y = x + 3.14**, where **x** is a tensor of arbitrary shape. If the length of **value** is 1, that single value is repeated, so **constant(shape = [1,3], value = [3.14])** is equivalent to **constant(shape = [1,3], value = [3.14, 3.14, 3.14])**.

### 4.1.3. Variables

A **variable** is a tensor that may be fed an initial value, may be updated by a computation, and its value persists between consecutive invocations of the computational graph. When a tensor is introduced as a variable, it is possible to update it later (see [Variable Updates](#)).

```

fragment variable<? = scalar>(
  Ê shape: integer[],          # the shape of the tensor
  Ê label: string )            # a label for referencing the tensor externally
-> ( output: tensor<?> )

```

### Argument validity

¥ Items in **shape** must be strictly positive.

¥ **label** must not be empty. Labels must only contain the following characters: **[a-z]**, **[A-Z]**, **[0-9]**, **\_**, **-**, **.**, **/**, **\**. If a variable operation has the same **label** as another variable operation in the graph (according to case insensitive comparison), then they share the underlying data, and they must have the same **shape**.

### Result semantics

¥ The shape of **output** is equal to **shape**. The rank of **output** is equal to the length of **shape**.

The content of **output** is not defined by the operation. The tensor may be filled with data from an external source and may be updated by **update** operations. The **label** argument is used to link the result to externally **stored** tensor data.

## 4.2. Element-wise Operations

Element-wise operations perform the same operation on each element of a tensor, irrespective of tensor dimensions and extents. The operations can be defined by simple mathematical formulae. In what follows, for the sake of simplicity, we will use one-dimensional indexing (index **i** runs through the whole tensor) and C style pseudo-code to define the result of each operation.

### 4.2.1. Unary Operations

Unary operations have a single tensor argument and return a single result.

Arithmetic operations:

```

fragment copy<?>( x: tensor<?> ) -> ( y: tensor<?> )
fragment neg( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment rcp( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment exp( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment log( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment abs( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment sign( x: tensor<scalar> ) -> ( y: tensor<scalar> )

```

Logical operations:

```

fragment not( x: tensor<logical> ) -> ( y: tensor<logical> )

```

Rounding operations:

```
fragment floor( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment ceil ( x: tensor<scalar> ) -> ( y: tensor<scalar> )
fragment round( x: tensor<scalar> ) -> ( y: tensor<scalar> )
```

### Result semantics

¥ The shape of **y** is equal to that of **x**.

For the operation **neg**, the output is defined as:

$$y[i] = -x[i]$$

For the operation **rcp**, the output is defined as:

$$y[i] = \frac{1}{x[i]}$$

For the operation **exp**, the output is defined as:

$$y[i] = e^{x[i]}$$

For the operation **log**, the output is defined as:

$$y[i] = \ln x[i]$$

For the operation **abs**, the output is defined as:

$$y[i] = \begin{cases} x[i] & \text{if } x[i] \geq 0 \\ -x[i] & \text{if } x[i] < 0 \end{cases}$$

For the operation **sign**, the output is defined as:

$$y[i] = \begin{cases} 1 & \text{if } x[i] > 0 \\ 0 & \text{if } x[i] = 0 \\ -1 & \text{if } x[i] < 0 \end{cases}$$

For the operation **floor**, the output is defined as:

$$y[i] = \text{floor}(x[i])$$

For the operation **ceil**, the output is defined as:

$$y[i] = \text{ceil}(x[i])$$

For the operation **round**, the output is defined as:

$$y[i] = \text{floor}(x[i] + 0.5)$$

## 4.2.2. Binary Operations

Binary operations take two tensor arguments and produce a single result. In the basic case of binary operations, by default, the shapes of both input tensors are the same, and the resulting output will also have the same shape. However, binary operations also support the case when the

either operand has singleton shape in a dimension but the other operand is non-singleton; in this case the value of the singleton dimension is repeated (broadcast) along that dimension (for example adding a column vector to all columns of a matrix). An extreme case is a scalar operand repeated in all dimensions.

Arithmetic operations:

```
fragment add( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
fragment sub( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
fragment mul( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
fragment div( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
fragment pow( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
```

Comparison operations:

```
fragment lt( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<logical> )
fragment gt( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<logical> )
fragment le( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<logical> )
fragment ge( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<logical> )
fragment eq( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<logical> )
fragment ne( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<logical> )
```

Logical operations:

```
fragment and( x: tensor<logical>, y: tensor<logical> ) -> ( z: tensor<logical> )
fragment or ( x: tensor<logical>, y: tensor<logical> ) -> ( z: tensor<logical> )
```

### Argument validity

- ¥ For each dimension, the extents of **x** and **y** must either be equal or one of them must be singular. Inference APIs are recommended to support the case when broadcasting happens in the *batch* and in the *spatial* dimensions (*channel* dimensions equal), and the case when broadcasting happens in all dimensions (one of the arguments is a scalar).

### Result semantics

- ¥ The rank of **z** is the maximum of the rank of **x** and **y**. For each dimension, if the extents of **x** and **y** are equal, the extent is carried over to **z**. Otherwise, the non-singular extent is carried over to **z**.
- ¥ The computations performed by these operators are as usual (see [Built-in Operators](#) for their mapping to mathematical operators).

## 4.2.3. Select Operation

The ternary operation **select** returns either of two values based on a condition (per element).

```

fragment select<?>(
  Ê condition: tensor<logical>,      # the condition for selecting the result
  Ê true_value: tensor<?>,          # the result when the condition is true
  Ê false_value: tensor<?> )        # the result when the condition is false
-> ( output: tensor<?> )

```

### Argument validity

¥ For each dimension, the extents of `condition`, `true_value` and `false_value` must either be equal or some of them must be singular.

### Result semantics

¥ For each dimension, if the extents of `condition`, `true_value` and `false_value` are equal, the extent is carried over to `output`. Otherwise, the non-singular extent is carried over to `output`.

The selection condition is evaluated independently for each entry (if `condition` shape is not singular). The `output` equals `true_value` where `condition` is `true` and `false_value` otherwise. Arguments of singular shape are broadcast to the shape of `output`.

As a special case, the condition may evaluate to `true` or `false` for all items (in runtime), in which case the whole subgraph calculating `true_value` or `false_value` can be omitted (in runtime), which provides a chance for optimization (conditional execution). This is especially useful if the shape of `condition` is singular in all dimensions. For example:

```

fragment calculate_condition( data: tensor<scalar> )
-> ( condition: tensor<logical> ) { ... }

fragment calculate_more_outputs( data: tensor<scalar> )
-> ( output: tensor<scalar> ) { ... }

data = ...
condition = calculate_condition(data)
output = select(condition, calculate_more_outputs(data), 0.0)

```

In the above example, if `condition` evaluates to `false`, the whole `calculate_more_outputs()` invocation can be omitted, which may contain an arbitrary large sub-graph.

!

Although the operation `select` and the `if-else` syntactic construct share similarities, they serve different purposes; the `select` operation serves *run-time* branching, while the `if-else` construct serves *compile-time* branching (in compositional syntax).

## 4.2.4. Simplifier Operations

Some convenience operations are provided for often-used element-wise operations that can be expressed via other primitives.

```
fragment sqr( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = pow(x, 2.0);
}
```

```
fragment sqrt( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = pow(x, 0.5);
}
```

```
fragment rsqr( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = pow(x, -2.0);
}
```

```
fragment rsqrt( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = pow(x, -0.5);
}
```

```
fragment log2( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = log(x) / log(2.0);
}
```

```
fragment min( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
{
    z = select(x < y, x, y);
}
```

```
fragment max( x: tensor<scalar>, y: tensor<scalar> ) -> ( z: tensor<scalar> )
{
    z = select(x > y, x, y);
}
```

```
fragment clamp( x: tensor<scalar>, a: tensor<scalar>, b: tensor<scalar> )
-> ( y: tensor<scalar> )
{
    y = max(min(x, b), a);
}
```

## 4.3. Sliding-Window Operations

Sliding-window operations come in pairs, a 'basic' version and a 'reverse' version (it could also be called forward and backward version, however, we would like to avoid confusion with the backward computation of back-propagation, where the backward computation of a 'reverse' operation may be a 'basic' operation). In general, the basic operations either keep the input shape or down-scale the input, while the reverse operations keep the input shape or up-scale it. Therefore, to clearly denote which shape we are talking about, we will denote them *down-scaled* and *up-scaled*, respectively. The basic operations map from the up-scaled shape to the down-scaled shape, and the reverse operations map from the down-scaled shape to the up-scaled shape.

Most of the parameters of sliding-window operations are common, thus they are summarized here, and for each operation only the additional parameters are described.

### Common Parameters

- ¥ **input**: `tensor<scalar>` : the tensor to be filtered.
- ¥ **output**: `tensor<scalar>` : the resulting tensor.
- ¥ **size**: `integer[]` : the shape of the kernel.
- ¥ **border**: `string` : the mode by which the borders are handled (see below).
- ¥ **padding**: `(integer, integer)[]` : the extents of the padding applied on the edges of the input. The values are supplied separately for each dimension and both sides (may be asymmetric). An empty array indicates that the padding extents are calculated automatically, see below.
- ¥ **stride**: `integer[]` : the amount with which the kernel is displaced in input space when moving the window.
- ¥ **dilation**: `integer[]` : the amount with which the kernel entries are displaced while matching the kernel to the input in a single input position. Dilation of 1 means that the kernel is continuously matched to the input (no gap).
- ¥ **output\_shape**: `integer[]` : the reference shape for reverse operations, typically the up-scaled shape of the input of a previous basic operation. An empty array indicates no reference shape, so the output shape is calculated as indicated below. If **output\_shape** is not empty, then applying the down-scaled shape calculation to **output\_shape** must result in the shape of **input**.

The parameters **size**, **padding**, **stride** and **dilation** must contain one entry for each relevant dimension. Alternatively, the arrays **padding**, **stride** and **dilation** may be empty (`[]`). If the arrays **stride** and **dilation** are empty, they are considered to be 1s in all relevant dimensions. If the array **padding** is empty, then padding is calculated automatically, see below.

The **padding**, **stride** and **dilation** parameters are always interpreted in the up-scaled space (the input space for the basic operations, the output space for reverse operations). The **border** parameter however, always effects the input values of each operation, regardless of whether it is basic or reverse operation.

### Argument Validity

- ¥ Items in **size** must be strictly positive. The number of required items depends on the operation.

- ¥ The number of required items in **padding** depends on the specific operation. Note, that the items in **padding** can be negative. Inference APIs are recommended to support the cases when all padding items are less in absolute value than the corresponding items in **size**.
- ¥ Items in **stride** must be strictly positive. The number of required items depends on the specific operation. Inference APIs are recommended to support the cases when all items are less or equal to the corresponding items in **size**.
- ¥ Items in **dilation** must be strictly positive. The number of required items depends on the specific operation. Inference APIs must support at least the case when all items are singular.
- ¥ **border** must be one of the modes specified **below**.

### Output Shape Calculation

For each dimension, the relation between up-scaled extent ( $X$ ), down-scaled extent ( $x$ ) filter **size** ( $f$ ), **padding** ( $p, q$ , where  $p$  corresponds to the padding that precedes, and  $q$  corresponds to the padding that succeeds the data), **stride** ( $s$ ) and **dilation** ( $d$ ) is as follows. Define dilated filter size as  $f_d = (f - 1) \cdot d + 1$ . Note, that when  $d = 1$  then  $f_d = f$ . Then the down-scaled extent is

$$x = \text{floor}\left(\frac{p + X + q - f_d}{s}\right) + 1.$$

The nominator must be non-negative, that is,  $p + X + q \geq f_d$ . Furthermore, the up-scaled extent is

$$X = (x - 1) \cdot s + f_d - (p + q).$$

Note, that because the down-scaling may involve rounding, the up-scaled extent is not necessarily equal to the one which it was down-scaled from.

### Automatic Padding Calculation

Padding is automatically calculated from the other parameters in all dimensions where it is not specified, such that  $x = \text{ceil}\left(\frac{X}{s}\right)$ . To achieve the desired down-scaled extent, define total padding as

$$t = (x - 1) \cdot s + f_d - X$$

and let the possibly asymmetric padding be defined as

$$p = \text{floor}\left(\frac{t}{2}\right) \quad (1)$$

$$q = \text{ceil}\left(\frac{t}{2}\right) \quad (2)$$

Note, that for all trailing dimensions, where  $f = 1$  and  $s = 1$  we have  $x = X$  and as a result  $p = 0$  and  $q = 0$ .

### Border Modes

The allowed border modes are as follows:

- ¥ **'ignore'**: the border values are not considered in the calculations (such as max or average)
- ¥ **'constant'**: the border is considered constant, currently the only supported value is 0
- ¥ **'replicate'**: the border is filled with the edge values replicated



¥ 'reflect': the values are taken from the input as if it was reflected to the edge (the edge is not duplicated)

¥ 'reflect-even': similar to 'reflect', but the edge is duplicated

Suppose for the sake of explanation that the input is one dimensional and has extent  $z$ . Let  $\tilde{\text{input}}$  denote the extended input, where the values outside the true input range are defined by the `border` parameter.

If `border = 'constant'`, we have:

$$\tilde{\text{input}}[i] = \begin{cases} c & \text{if } i < 0 \\ \text{input}[i] & \text{if } 0 \leq i < z \\ c & \text{if } i \geq z \end{cases}$$

For sliding-window operations, only the case of  $c = 0$  is defined.

If `border = 'replicate'`, we have:

$$\tilde{\text{input}}[i] = \begin{cases} \text{input}[0] & \text{if } i < 0 \\ \text{input}[i] & \text{if } 0 \leq i < z \\ \text{input}[z-1] & \text{if } i \geq z \end{cases}$$

If `border = 'reflect'`, we have:

$$\tilde{\text{input}}[i] = \begin{cases} \text{input}[-i] & \text{if } i < 0 \\ \text{input}[i] & \text{if } 0 \leq i < z \\ \text{input}[z-1-(i-z+1)] & \text{if } i \geq z \end{cases}$$

If `border = 'reflect-even'`, we have:

$$\tilde{\text{input}}[i] = \begin{cases} \text{input}[-i-1] & \text{if } i < 0 \\ \text{input}[i] & \text{if } 0 \leq i < z \\ \text{input}[z-1-(i-z)] & \text{if } i \geq z \end{cases}$$

!

Dimensionality of operations is not explicitly denoted, instead, it follows from the dimensions of the arguments. Let the padded input shape in each dimension be defined as  $p + X + q$ . If the padded input has rank  $n$ , then the operation is  $n$ -dimensional, and has  $n-2$  *spatial* dimensions.

### 4.3.1. Convolution and Deconvolution

The `conv` operation correlates a filter with an input, while the `deconv` operation performs the reverse, which is roughly equivalent to mathematical convolution. Up and down-scaling of tensor shapes happens only in the *spatial* dimensions. The *batch* dimension is the same for inputs and outputs, while the *channel* dimension of the output is derived from the *batch* dimension of the filters.

```

fragment conv(
  Ê input: tensor<scalar>,
  Ê filter: tensor<scalar>,          # the filter that the input is convolved with
  Ê bias: tensor<scalar> = 0.0,      # the bias that is added to the output
  Ê border: string = 'constant',
  Ê padding: (integer,integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [],
  Ê groups: integer = 1 )           # the number of convolution groups
-> ( output: tensor<scalar> )

fragment deconv(
  Ê input: tensor<scalar>,
  Ê filter: tensor<scalar>,
  Ê bias: tensor<scalar> = 0.0,
  Ê border: string = 'constant',
  Ê padding: (integer,integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [],
  Ê output_shape: integer[] = [],
  Ê groups: integer = 1 )
-> ( output: tensor<scalar> )

```

### Argument validity

- ¥ The rank of `filter` must equal to that of `input`.
- ¥ The role of the common parameter `size` is performed by the shape of `filter`, therefore the general validity constraints defined above for `size` apply for the shape of `filter`. Inference APIs are recommended to support cases where the filter's *spatial* dimensions are between 1 and 5, inclusive.
- ¥ The length of `padding`, `stride` and `dilation` must equal the number of spatial dimensions or 0. Item  $i$  in `padding`, `stride` and `dilation` corresponds to *spatial* dimension  $i$ . Inference APIs are recommended to support cases where the `stride` is 1 or 2.
- ¥ `groups` must be non-negative, and if positive, it must be a divisor of the *batch* dimension of the shape of `filter`. The special value 0 indicates *depth-wise separable convolution*.
- ¥ The *channel* dimension of the `filter` times the number of `groups` must equal the *channel* dimension of the `input`.
- ¥ The *channel* dimension of `bias` must equal the *batch* dimension of `filter` or must be singular. In all other dimensions, `bias` must have singular shape.
- ¥ `border` must be one of 'constant', 'replicate', 'reflect', 'reflect-even'. Inference APIs are recommended to support cases where `border` is 'constant'.
- ¥ The length of `output_shape` must equal the rank of `input` or 0.

### Result semantics

- ¥ The rank of `output` is the same as the rank of `input`.

- ¥ The extent of the *batch* dimension of the shape of **output** equals that of the **input**.
- ¥ The extent of the *channel* dimension of the shape of **output** equals the extent of the *batch* dimension of the shape of **filter**.
- ¥ For each other dimension, the shape of **output** is calculated as defined **above** for sliding window operations ( $x$  in case of **conv** and  $X$  in case of **deconv**).

Operation **conv** maps from the up-sampled space to the down-sampled space, while **deconv** maps from the down-sampled space to the up-sampled space.

For the sake of explanation, suppose that the **input** and **filter** tensors are one dimensional (there is only one *spatial* dimension, *batch* and *channel* dimensions are singular). Furthermore, let **bias** be omitted (as if it was zero). Then for each index  $i \in [0, x]$ , the output of **conv** is calculated as (mathematical correlation):

$$\text{output}[i] = \sum_{j=0}^{f-1} \tilde{\text{input}}[i \cdot s + j \cdot d - p] \cdot \text{filter}[j]$$

The **deconv** operation implements a calculation as if it was the reverse of correlation (i.e. mathematical convolution), taking stride and dilation into account. For each index  $i \in [0, X]$ , the output of **deconv** is calculated as:

$$\text{output}[i] = \sum_{j=0}^{f-1} \tilde{\text{input}}[(i + p - j \cdot d) / s] \cdot \text{filter}[j]$$

where the sum only includes terms for which  $(i + p - j \cdot d) \bmod s = 0$ .

In general, the up-scaled space has dimensions  $(B, C, X_1, X_2, \dots)$ , the down-scaled space has shape  $(B, c, x_1, x_2, \dots)$ , and the filter has dimensions  $(c, C, f_1, f_2, \dots)$ . The following equations will suppose two *spatial* dimensions, but generalization to more dimensions is straightforward.

In case of the **conv** operation, for each batch index  $b \in [0..B)$  and for each  $k_2 \in [0..c)$ , the output is calculated as:

$$\text{output}[b][k_2][i_1][i_2] = \sum_{k_1=0}^{C-1} \sum_{j_1=0}^{f_1-1} \sum_{j_2=0}^{f_2-1} \tilde{\text{input}}[b][k_1][i_1 \cdot s_1 + j_1 \cdot d_1 - p_1][i_2 \cdot s_2 + j_2 \cdot d_2 - p_2] \cdot \text{filter}[k_2][k_1][j_1][j_2]$$

and in case of **deconv**, for each  $k_1 \in [0..C)$ :

$$\text{output}[b][k_1][i_1][i_2] = \sum_{k_2=0}^{c-1} \sum_{j_1=0}^{f_1-1} \sum_{j_2=0}^{f_2-1} \tilde{\text{input}}[b][k_2][(i_1 + p_1 - j_1 \cdot d_1) / s_1][(i_2 + p_2 - j_2 \cdot d_2) / s_2] \cdot \text{filter}[k_2][k_1][j_1][j_2]$$

When taking **bias** into account, the output is defined as if **bias** was added to the result of convolution without bias:

$$\text{conv}(\text{input}, \text{filter}, \text{bias}, \dots) = \text{conv}(\text{input}, \text{filter}, \text{bias} = 0.0, \dots) + \text{bias}$$

## Grouped Convolutions

In case of grouped convolution/deconvolution (**groups** > 1), conceptually, both the **input** and **output**

are split into **groups** number of segments along the *channel* dimension, and the operations are executed independently on the split segments. Note, that in this case, the *channel* dimension of the **filter** must be just the right extent to match one segment's *channel* dimension. Letting  $G$  denote the number of groups, the **filter** must be of shape  $(c, C / G, f_1, f_2, \dots)$ . Note, that the number of output channels *per group* will be  $c / G$ .

A special case of grouped convolution is when  $C = G$ , in which case the convolution is performed independently for each channel (also called depth-wise separable or plane-wise). The ratio  $c / G$  is also known as *channel multiplier* in this case. The special value **groups** = 0 is a shorthand for specifying depth-wise separable convolution without requiring to reference the actual number of channels.

### 4.3.2. Box Filter

Box filtering performs summation in a local window.

```
fragment box(
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer, integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [],
  Ê normalize: logical = false )      # whether to normalize by the kernel volume
-> ( output: tensor<scalar> )

fragment debox(
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer, integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [],
  Ê output_shape: integer[] = [],
  Ê normalize: logical = false )
-> ( output: tensor<scalar> )
```

#### Argument validity

- ¥ Inference APIs are recommended to support cases where **size** is between 1 and 3, inclusive.
- ¥ The length of **padding**, **stride** and **dilation** must equal the rank of **input** or 0. Item  $i$  in **padding**, **stride** and **dilation** corresponds to dimension  $i$ . Inference APIs are recommended to support cases where **stride** is 1 or 2.
- ¥ **border** must be one of 'ignore', 'constant', 'replicate', 'reflect', 'reflect-even'. Inference APIs are recommended to support the 'constant' value.
- ¥ The length of **output\_shape** must equal the rank of **input** or 0.

#### Result semantics

- ¥ The rank of **output** is the same as the rank of **input**. For each dimension, the shape of **output** is calculated as defined **above** for sliding window operations ( $x$  in case of **box** and  $X$  in case of **debox**).
- ¥ When **normalize = true**, the output is divided with the volume of the kernel (as indicated by the **size** parameter).

Box filtering in *spatial* dimensions is equivalent to (depth-wise separable) convolution with a weight tensor of all 1s. Note however, that box filtering does not differentiate *batch* and *channel* dimensions, and it may also be applied to those dimensions.

For the sake of explanation, suppose that the **input** tensor is one dimensional. Then in the unnormalized case, for each index  $i \in [0, x)$ , the output of **box** is calculated as:

$$\text{output}[i] = \sum_{j=0}^{f-1} \text{input}[i \cdot s + j \cdot d - p]$$

The **debox** operation implements a calculation as if it was the reverse of **box**, taking stride and dilation into account. For each index  $i \in [0, X)$ , the output of **debox** is calculated as:

$$\text{output}[i] = \sum_{j=0}^{f-1} \text{input}[(i + p - j \cdot d) / s]$$

where the sum only includes terms for which  $(i + p - j \cdot d) \bmod s = 0$ .

The box filter is separable in the dimensions, so box filtering in multiple dimensions is equivalent to a sequence of 1-dimensional box filters.

### 4.3.3. Index Based Sampling

The operation **argmax\_pool** returns the maximum value positions in the local window.

```
fragment argmax_pool (
  Ê input: tensor,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer, integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [] )
-> ( index: tensor<integer> )           # the positions where the maxima are found
```

#### Argument validity

- ¥ Inference APIs are recommended to support cases where **size** is between 1 and 3, inclusive.
- ¥ The length of **padding**, **stride** and **dilation** must equal the rank of **input** or 0. Item  $i$  in **padding**, **stride** and **dilation** corresponds to dimension  $i$ . Inference APIs are recommended to support cases where **stride** is 1 or 2.
- ¥ **border** must be one of 'ignore', 'constant', 'replicate', 'reflect', 'reflect-even'. Inference APIs are recommended to support 'constant' and 'ignore' values.

## Result semantics

¥ The rank of **output** is the same as the rank of **input**. For each dimension, the shape of **index** is calculated as defined [above](#) for sliding window operations ( $x$ ).

For each position in the output space, **argmax\_pool** chooses the maximum value in the kernel window, and outputs its index within the window. Indices correspond to row-major ordering of positions within the window. For the sake of explanation, suppose that the **input** tensor is one dimensional. Then for each index  $i \in [0, x)$ , the output of **argmax\_pool** is calculated as:

$$\text{index}[i] = \underset{j=0}{\overset{f-1}{\text{argmax}}} \text{input}[i \cdot s + j \cdot d - p]$$

If there are multiple equal maximal values, only the first index is stored in **index**. Note, that the **argmax** operation is not separable.

Two related operations are **sample** and **desample**. The operation **sample** performs sampling given the indices, obtained from the **argmax\_pool** operation. That is, it fetches the actual values indicated by the indices. The operation **desample** performs the reverse, populating the up-scaled space from the values in the down-scaled space using the sampling indices, summing values that are mapped to the same up-scaled position.

```
fragment sample(  
  Ê input: tensor<scalar>,  
  Ê index: tensor<integer>,          # the positions where values are sampled from  
  Ê size: integer[],  
  Ê border: string = 'constant',  
  Ê padding: (integer, integer)[] = [],  
  Ê stride: integer[] = [],  
  Ê dilation: integer[] = [] )  
-> ( output: tensor<scalar> )  
  
fragment desample(  
  Ê input: tensor<scalar>,  
  Ê index: tensor<integer>,          # the positions where values are sampled to  
  Ê size: integer[],  
  Ê border: string = 'constant',  
  Ê padding: (integer, integer)[] = [],  
  Ê stride: integer[] = [],  
  Ê dilation: integer[] = [],  
  Ê output_shape: integer[] = [] )  
-> ( output: tensor<scalar> )
```

## Argument validity

¥ Inference APIs are recommended to support cases where **size** is between 1 and 3, inclusive.

¥ The shape of **index** must be the same as that of **output** for **sample** and that of **input** for **desample**.

¥ The length of **padding**, **stride** and **dilation** must equal the rank of **input** or 0. Item  $i$  in **padding**,

`stride` and `dilation` corresponds to dimension  $i$ . Inference APIs are recommended to support cases where `stride` is 1 or 2.

¥ `border` must be one of 'constant', 'replicate', 'reflect', 'reflect-even' in case of `sample` and must be 'constant' in case of `desample`. Inference APIs are recommended to support the 'constant' value.

¥ The length of `output_shape` must equal the rank of `input` or 0.

### Result semantics

¥ The rank of `output` is the same as the rank of `input`. For each dimension, the shape of `output` is calculated as defined above for sliding window operations ( $x$  in case of `sample` and  $X$  in case of `desample`).

For the sake of explanation, suppose that the `input` and `index` tensors are one dimensional. Then for each index  $i \in [0, x)$ , the output of `sample` is calculated as:

$$\text{output}[i] = \tilde{\text{input}}[i \cdot s + \text{index}[i] \cdot d - p]$$

For each index  $i \in [0, X)$ , the output of `desample` is calculated as:

$$\text{output}[i] = \sum_{\substack{j=0 \\ 0 \leq (i+p-j \cdot d) / s < x \\ \text{index}[(i+p-j \cdot d) / s] = j}}^{f-1} \text{input}[(i+p-j \cdot d) / s]$$

where the sum only includes terms for which  $(i+p-j \cdot d) \bmod s = 0$ . Note, that any `border` value other than 'constant' (zeros) loses its utility because the condition  $\text{index}[(i+p-j \cdot d) / s] = j$  on the sampling indices is never met outside the valid region.

### 4.3.4. Up and Down-Sampling

Tensors may be up and down-sampled along the *spatial* dimensions. For down-sampling, two methods are given: area interpolation and nearest neighbor interpolation. For up-sampling, also two methods are given: multi-linear interpolation and nearest neighbor interpolation. Only integer scaling factors are allowed, such that these methods are consistent with other sliding window operations. Some of these operations are readily expressible via other primitives.

The two down-sampling methods are:

```
fragment nearest_downsample( input: tensor<scalar>, factor: integer[] )
-> ( output: tensor<scalar> )
{
    Ê dims = 2 + length_of(factor);      # +2 for batch and channel dimensions
    Ê output = box(input, size = [1] * dims, stride = [1,1] + factor,
    Ê         padding = [(0,0)] * dims);
}
```

```

fragment area_downsample( input: tensor<scalar>, factor: integer[] )
-> ( output: tensor<scalar> )
{
    Ê   dims = 2 + length_of(factor);          # +2 for batch and channel dimensions
    Ê   output = box(input, size = [1,1] + factor, stride = [1,1] + factor,
    Ê           padding = [(0,0)] * dims, normalize = true);
}

```

The two up-sampling methods are:

```

fragment nearest_upsample( input: tensor<scalar>, factor: integer[] )
-> ( output: tensor<scalar> )
{
    Ê   dims = 2 + length_of(factor);          # +2 for batch and channel dimensions
    Ê   output = debox(input, size = [1,1] + factor, stride = [1,1] + factor,
    Ê           padding = [(0,0)] * dims);
}

```

```

fragment multilinear_upsample(
    Ê   input: tensor<scalar>,                # the tensor to up-sample
    Ê   factor: integer[],                    # the up-sampling factor
    Ê   method: string = 'symmetric',         # the linear interpolation method to use
    Ê   border: string = 'replicate' )
-> ( output: tensor<scalar> )                # the up-sampled tensor

```

### Argument validity

- ¥ The items in **factor** must be strictly positive. Item  $i$  corresponds to *spatial* dimension  $i$ . Inference APIs are recommended to support the case when **factor** is 2.
- ¥ **method** must be one of 'asymmetric', 'symmetric', 'aligned' (see below). Inference APIs are recommended to support one of values 'asymmetric' or 'symmetric'.
- ¥ **border** must be one of 'constant', 'replicate', 'reflect', 'reflect-even'. Inference APIs are recommended to support the value 'constant'.

### Result semantics

- ¥ The rank of **output** is the same as the rank of **input**.
- ¥ For the *batch* and *channel* dimensions, the shape of **output** is the same as the shape of **input**.
- ¥ For each *spatial* dimension, the output shape is calculated as follows. Let  $f$  be the corresponding sampling factor. For down-sampling,  $x = \frac{X}{f}$ , where the division must result in an integer without modulo. For up-sampling,  $X = x \cdot f$ .

Multi-linear up-sampling weighs input values to form an output value. In a single dimension, the weighting happens according to one of various schemes. Let  $f$  be the sampling factor, and let us assume that the input is one dimensional. In the basic scheme, for output index  $i$ , let  $j(i) = \text{floor}(\frac{i}{f})$



and let  $u(i) = \text{frac}\left(\frac{i}{f}\right)$ . Then the output is calculated as:

$$\text{output}[i] = (1 - u(i)) \cdot \text{input}[j(i)] + u(i) \cdot \text{input}[j(i) + 1]$$

When  $j(i)+1$  equals the input extent  $x$ , the index  $j(i)+1$  is clamped to  $x-1$ , so index  $j(i)$  is used, which results in  $\text{output}[i] = \text{input}[j(i)]$  for multiple values of  $i$  on the higher end of the output range. For this reason, this method is called '**asymmetric**'.

To correct this asymmetric edge effect, various methods exist. One is to change the weighting such that the indices need not be clamped, by aligning the edges of the input and output intervals. Define  $g(x) = \frac{x \cdot f - 1}{x - 1}$ , and define  $j(i) = \text{floor}\left(\frac{i}{g(x)}\right)$  and  $u(i) = \text{frac}\left(\frac{i}{g(x)}\right)$ . This way, only the last point of the output interval maps to the last point in the input interval, hence the method is called '**aligned**'. A potential disadvantage of this method is that the weighting depends on the input extent  $x$ .

An alternative solution is to make the edge effect symmetric, by shifting the index calculation, resulting in the method called '**symmetric**'. Let  $f_0 = \frac{f-1}{2}$ , and define  $j(i) = \text{floor}\left(\frac{i-f_0}{f}\right)$  and  $u(i) = \text{frac}\left(\frac{i-f_0}{f}\right)$ . When  $j = -1$  or  $j+1 = x$ , the index is clamped to 0 or  $x-1$  respectively.

Clamping along the edges corresponds to **border** = '**replicate**'. Alternatively, the values outside the input range may be taken as zeros, which corresponds to **border** = '**constant**'.

When **method** = '**symmetric**' or **method** = '**asymmetric**', multilinear upsampling can be described as a (depth-wise separable) deconvolution with constant weights that only depend on the upsampling factor, and are uniform in the whole input range. As special cases, here are the equivalents for the linear and bilinear case when the sampling factor is uniformly 2. Note, that these definitions do not form part of the standard, they are only shown here for convenience.

```
fragment _linear_upsample2x_symmetric( input: tensor<scalar>, border: string )
-> ( output: tensor<scalar> )
{
    channels = shape_of(input)[1];
    filter = constant(shape = [channels, 1, 4],
                     value = [0.25, 0.75, 0.75, 0.25] * channels);
    output = deconv(input, filter, stride = [2], padding = [(1,1)],
                   border = border, groups = channels);
}
```

```
fragment _linear_upsample2x_asymmetric( input: tensor<scalar>, border: string )
-> ( output: tensor<scalar> )
{
    channels = shape_of(input)[1];
    filter = constant(shape = [channels, 1, 3],
                     value = [0.5, 1.0, 0.5] * channels);
    output = deconv(input, filter, stride = [2], padding = [(0,1)],
                   border = border, groups = channels);
}
```

```

fragment _bilinear_upsample2x_symmetric( input: tensor<scalar>, border: string )
-> ( output: tensor<scalar> )
{
  channels = shape_of(input)[1];
  weights = [0.0625, 0.1875, 0.1875, 0.0625,
             0.1875, 0.5625, 0.5625, 0.1875,
             0.1875, 0.5625, 0.5625, 0.1875,
             0.0625, 0.1875, 0.1875, 0.0625];
  filter = constant(shape = [channels, 1, 4, 4], value = weights * channels);
  output = deconv(input, filter, stride = [2,2], padding = [(1,1), (1,1)],
                  border = border, groups = channels);
}

```

```

fragment _bilinear_upsample2x_asymmetric( input: tensor<scalar>, border: string )
-> ( output: tensor<scalar> )
{
  channels = shape_of(input)[1];
  weights = [0.25, 0.5, 0.25,
             0.50, 1.0, 0.50,
             0.25, 0.5, 0.25];
  filter = constant(shape = [channels, 1, 3, 3], value = weights * channels);
  output = deconv(input, filter, stride = [2,2], padding = [(0,1), (0,1)],
                  border = border, groups = channels);
}

```

## 4.4. Reduce Operations

Reduce operations result in a tensor whose shape is singular along those axes where the input is reduced.

```

fragment sum_reduce(
  input: tensor<scalar>,          # the tensor to be reduced
  axes: integer[],                # the axes along which to reduce
  normalize: logical = false )   # whether to normalize with the reduction volume
-> ( output: tensor<scalar> )     # the reduced tensor

```

```

fragment max_reduce(
  input: tensor<scalar>,
  axes: integer[] )
-> ( output: tensor<scalar> )

```

```

fragment min_reduce(
    Ê input: tensor<scalar>,
    Ê axes: integer[] )
-> ( output: tensor<scalar> )

```

```

fragment argmax_reduce(
    Ê input: tensor<scalar>,
    Ê axes: integer[] )
-> ( output: tensor<integer> )

```

```

fragment argmin_reduce(
    Ê input: tensor<scalar>,
    Ê axes: integer[] )
-> ( output: tensor<integer> )

```

### Argument validity

¥ Items in **axes** must be unique, non-negative and less than the rank of **input**. Inference APIs are recommended to support the case when the **axes** parameter includes all dimensions except the *channel* dimension.

### Result semantics

¥ The rank of **output** is the same as the rank of **input**. The shape of **output** is singleton along the dimensions listed in **axes** and is the same as the shape of **input** in all other dimensions.

For the sake of argument, reduction operations are described for a single axis. Since these reductions are separable, for multiple axes, it is equivalent to a sequence of single-axis reductions. Furthermore, for the sake of explanation, let the input be of shape  $(m, n)$ , and we perform the reduction on the second axis, and the result becomes of shape  $(m, 1)$ . For each index  $i \in [0, m)$ , for **sum\_reduce** we have

$$\text{output}[i, 0] = \sum_{j=0}^{n-1} \text{input}[i, j]$$

for **max\_reduce** we have

$$\text{output}[i, 0] = \max_{j=0}^{n-1} \text{input}[i, j]$$

for **min\_reduce** we have

$$\text{output}[i, 0] = \min_{j=0}^{n-1} \text{input}[i, j]$$

for **argmax\_reduce** we have

$$\text{output}[i, 0] = \arg\max_{j=0}^{n-1} \text{input}[i, j]$$

for `argmin_reduce` we have

$$\text{output}[i, 0] = \arg\min_{j=0}^{n-1} \text{input}[i, j]$$

If `normalize = true` for `sum_reduce`, the sum is normalized by the volume of the `input` along the reduction axes (the product of the extents along the dimensions listed in `axes`), resulting in averaging (taking the mean). A convenience operation is provided for mean reduction:

```
fragment mean_reduce( input: tensor<scalar>, axes: integer[] )
-> ( output: tensor<scalar> )
{
    output = sum_reduce(input, axes = axes, normalize = true);
}
```

## 4.5. Tensor Shape Operations

The role of tensor shape manipulating operations is to prepare tensors to serve as parameters to operations that actually perform computations.

### 4.5.1. Reshaping

```
fragment reshape<?>(
    input: tensor<?>,          # the tensor to be reshaped
    shape: integer[] )         # the new shape
-> ( output: tensor<?> )       # the reshaped tensor
```

#### *Argument validity*

¥ Items in the `shape` array must all be non-negative or it must contain at most one `-1` item. If all items are positive, the product of items in `shape` must equal the volume of `input`. A `0` item means that the corresponding extent is inherited from the input. In case `shape` contains a `-1` item, the product of the positive and inherited items must divide the input volume without remainder, and the `-1` item is replaced with the quotient of the volume of `input` and the product, therefore meeting the equality constraint.

#### *Result semantics*

¥ The rank of `output` equals the length of `shape`. The shape of `output` equals `shape` (after substituting the `0` and `-1` items, if any).

The reshape operation considers the content of the tensor to be laid out linearly in row-major order. Reshaping does not affect the content of the tensor, instead it only provides a new indexing to the same data. For this reason, the amount of data indexed must be the same under the two (original and new) indexing schemes.

An often used case of reshaping is when singleton dimensions are removed or inserted, named `squeeze` and `unsqueeze` respectively. Specialized fragments for these cases avoid mentioning the explicit shape of tensors:

```
fragment squeeze<?>( input: tensor<?>, axes: integer[] ) -> ( output: tensor<?> )
fragment unsqueeze<?>( input: tensor<?>, axes: integer[] ) -> ( output: tensor<?> )
```

The `axes` parameter denotes the list of dimension indices to be removed in case of `squeeze` and in case of `unsqueeze`, `axes` denote the dimension indices as they will appear in the output after insertion.

!

For operation `squeeze`, the rank of `output` is the rank of `input` minus the length of `axes`. Items in `axes` must be non-negative and less than the rank of `input`.

For operation `unsqueeze`, the rank of the `output` is the rank of `input` plus the length of `axes`. Items in `axes` must be non-negative and less than the rank of `output`.

## 4.5.2. Transposing

```
fragment transpose<?>(
  Ê input: tensor<?>,          # the tensor to be transposed
  Ê axes: integer[] )          # the resulting order of axes
-> ( output: tensor<?> )      # the transposed tensor
```

### Argument validity

¥ If the `axes` array has  $n$  items, it must be a permutation of the integers from 0 to  $n - 1$  (inclusive). The length of `axes` must be less than or equal to the rank of `input`.

### Result semantics

¥ The rank of `output` is the same as the rank of `input`. Let the shape of `input` along the first  $n$  dimensions be  $(x_0, x_1, \dots, x_{n-1})$ . Then the shape of `output` along the first  $n$  dimensions equals  $(x_{axes[0]}, x_{axes[1]}, \dots, x_{axes[n-1]})$ . For the rest of the trailing dimensions, the shape of `output` is the same as the shape of `input`.

Transposing a tensor reorganizes its data. It is the  $n$  dimensional generalization of matrix transpose, allowing a permutation of the dimensions, and hence changing the ordering of the data.

Let  $k \in [0..n)$  and let  $x_k$  denote the extent of the `input` in dimension  $k$ . In  $n$ -dimensional indexing notation, where index  $i_k \in [0..x_k)$ , the output can be described as:

$$\text{output}[i_{axes[0]}, \dots, i_{axes[n-1]}] = \text{input}[i_0, \dots, i_{n-1}]$$

## 4.5.3. Splitting and Concatenation

Splitting and concatenation convert between a single tensor and sections of it, separated along one of the dimensions. That is, they convert between a (conceptually) single chunk of data and many

chunks of data.

```
fragment split<?>(  
  Ê value: tensor<?>,           # the tensor holding the data in one piece  
  Ê axis: integer,               # the dimension along which to split  
  Ê ratios: integer[]           # the ratios of the chunks along the split axis  
  -> ( values: tensor<?>[] )    # the list of tensors holding the data in chunks
```

### Argument validity

- ¥ Items in **ratios** must be positive.
- ¥ The sum of **ratios** must divide the extent of **value** along dimension **axis** without remainder.
- ¥ **axis** must be non-negative and less than the rank of **value**. Inference APIs are recommended to support the case when **axis** equals 1 (*channel* dimension).

### Result semantics

Let  $n$  be the number of items in **ratios**,  $r_i$  equal **ratios**[ $i$ ], and  $R = \sum_{i=0}^n r_i$ . Furthermore, let  $S$  equal the shape of **value** along dimension **axis**,  $\mu = S / R$ , and let  $s_i = \mu \cdot r_i$

- ¥ The number of items in **values** is equal to the number of items in **ratios**.
- ¥ The rank of **values**[ $i$ ] is the same as the rank of **value**.
- ¥ The shape of **values**[ $i$ ] along dimension **axis** equals  $s_i$ .
- ¥ The shape of **values**[ $i$ ] along all other dimensions equals the shape of **value**.

```
fragment concat<?>(  
  Ê values: tensor<?>[],        # the list of tensors holding the data in chunks  
  Ê axis: integer )              # the dimension along which to concatenate  
  -> ( value: tensor<?> )       # the tensor holding the data in one piece
```

### Argument validity

- ¥ The rank of **values**[ $i$ ] must be the same.
- ¥ The shapes of **values**[ $i$ ] for all dimensions other than **axis** must be the same.
- ¥ **axis** must be non-negative and less than the rank of **values**[ $i$ ]. Inference APIs are recommended to support the case when **axis** equals 1 (*channel* dimension).

### Result semantics

- ¥ The rank of **value** is the same as the rank of **values**[ $i$ ].
- ¥ The shape of **value** along dimension **axis** is equal to the sum of shapes of **values**[ $i$ ] along dimension **axis**.
- ¥ The shape of **value** along all other dimensions is equal to the shape of **values**[ $i$ ].

To precisely define the relation between the split **values** and the concatenated **value** for both **split** and **concat** operations, let  $\sigma_0 = 0$  and for  $i \in [1..n]$ , let  $\sigma_i = \sum_{j=0}^{i-1} s_j$ . For the sake of argument, suppose that **value** and **values[i]** are of rank 3 and **axis** = 1. Then, the values in **values[i]** are equal to the values in the section of **value** that runs from  $\sigma_i$  to  $\sigma_{i+1}$  along dimension **axis**:

$$\text{values}[i] = \text{value}[:, \sigma_i : \sigma_{i+1}, :]$$

where  $\sigma_i : \sigma_{i+1}$  means the section of the tensor from index  $\sigma_i$  (inclusive) to index  $\sigma_{i+1}$  (exclusive) along a given dimension, and the sole  $:$  means the whole section of the tensor along the given dimension.

A special case of concatenation and splitting is when an array of tensors are concatenated/split along a new (singular) dimension. These operations are called stacking and unstacking:

```
fragment stack<?>( values: tensor<?>[], axis: integer ) -> ( value: tensor<?> )
fragment unstack<?>( value: tensor<?>, axis: integer ) -> ( values: tensor<?>[] )
```

Stacking is conceptually equivalent to inserting a singleton dimension to each tensor at position given by **axis**, and then concatenating them along the axis. Unstacking is its reverse, splitting along the given axis to singleton slices, and then removing the singleton dimension.

!

For operation **stack**, the rank of **output** is the rank of **input** plus 1. The **axis** must be non-negative and less than the rank of **output**.

For operation **unstack**, the rank of **output** is the rank of **input** minus 1. The **axis** must be non-negative and less than the rank of **input**.

#### 4.5.4. Slicing

Slicing takes a section of a tensor along some specified dimensions.

```
fragment slice<?>(
    Ê input: tensor<?>,          # the tensor to be sliced
    Ê axes: integer[],           # axes along which to slice
    Ê begin: integer[],          # beginning of slice along each axis
    Ê end: integer[] )          # and of slice along each axis
-> ( output: tensor<?> )
```

##### *Argument validity*

- ¥ The length of arrays **axes**, **begin** and **end** must be the same.
- ¥ Items in **axes** must be non negative and less than the rank of **input**.
- ¥ For each item **axes[i]**, the items **begin[i]** and **end[i]** must be in the range  $(-x_i, x_i]$ , where  $x_i$  is the extent of **input** along the dimension **axes[i]**, and **end[i]** must be greater than **begin[i]**. If an item is negative, it is understood as if  $x_i$  was added to it, hence allowing indexing from the end.

The special value `end[i] == 0` means taking the range until  $x_i$  along dimension `axes[i]`.

### Result semantics

¥ The rank of `output` is the same as the rank of `input`. The shape of `output` along dimension  $j$  is `end[i] - begin[i]` (after accounting for non-positive values) if  $j == \text{axes}[i]$ . Otherwise, the shape of `output` along dimension  $j$  is the same as that of `input`.

For the sake of explanation, let the input be 3 dimensional, and let `axes = [1]`. Supposing positive values in `begin` and `end`, the output is calculated as:

```
output = input[:, begin[0]:end[0], :]
```

## 4.6. Region-of-Interest Operations

RoI operations crop regions of interest out of a feature map and pool them to a fixed size. The actual pooling operation may vary just as in case of regular pooling.

### 4.6.1. RoI Pooling

RoI pooling generates a fixed size output by pooling regions of variable size.

```
fragment avg_roi_pool (
  Ê input: tensor<scalar>,          # the feature maps to pool from
  Ê rois: tensor<scalar>,           # the regions of interest
  Ê batch_index: tensor<integer>,    # batch indices for each RoI
  Ê output_size: integer[] )        # the desired output size
-> ( output: tensor<scalar> )
```

```
fragment max_roi_pool (
  Ê input: tensor<scalar>,
  Ê rois: tensor<scalar>,
  Ê batch_index: tensor<integer>,
  Ê output_size: integer[] )
-> ( output: tensor<scalar> )
```

### Argument validity

Let  $N$  be the number of RoIs.

¥ `rois` must be a tensor of shape  $(N, 2 \cdot M)$ , where  $M$  is the number of spatial dimensions.

¥ `batch_index` must be a tensor of shape  $(N, 1)$ .

¥ `output_size` must have an entry for each spatial dimension. The items in `output_size` must be positive.

### Result semantics

Let the shape of `input` be  $(B, C, H, W)$ .



¥ The rank of **output** is the same as the rank of **input**. The shape of **output** is  $(N, C, p_H, p_W)$  where  $\text{output\_size} = (p_H, p_W)$ .

RoI coordinates are first rounded to the nearest integer values. Then, each RoI is cropped out from the feature map of the batch item indicated by the corresponding **batch\_index**.

Let a row in the **rois** tensor be denoted by a 4-tuple  $(y_0, x_0, y_1, x_1)$ . Let the height of the RoI be denoted as  $r_H = y_1 - y_0$ , and the width of a RoI be denoted as  $r_W = x_1 - x_0$ .

Each RoI is divided into  $p_H \times p_W$  cells. Cells have height  $c_H = \text{ceil}(\frac{r_H}{p_H})$  and width  $c_W = \text{ceil}(\frac{r_W}{p_W})$ , and are strided with strides  $s_H = \text{floor}(\frac{r_H}{p_H})$  and  $s_W = \text{floor}(\frac{r_W}{p_W})$ ; cell  $(i, j)$  starts at position  $(c_H + i * s_H, c_W + j * s_W)$ , where  $i$  and  $j$  are zero-based indices.

For each cell, the average or max values are calculated the same way as for the **avg\_pool** and **max\_pool** operations.

#### 4.6.2. RoI Align

Enhanced versions of RoI pooling aim to eliminate the quantization effect of rounding RoI coordinates to integer values and the way cells are generated. Instead, each RoI is directly mapped to a fixed intermediate size by resampling. Afterwards, max or average pooling can be applied to arrive to a final size.

The **roi\_resample** operation resamples a RoI to a fixed size using multilinear interpolation.

```
fragment roi_resample(
  Ê input: tensor<scalar>,           # the feature maps to crop from
  Ê rois: tensor<scalar>,            # the regions of interest
  Ê batch_index: tensor<integer>,    # batch indices for each RoI
  Ê output_size: integer[],          # the desired output size
  Ê method: string = 'symmetric' )   # interpolation method
-> ( output: tensor<scalar> )
```

##### Argument validity

Let  $N$  be the number of RoIs.

¥ **rois** must be a tensor of shape  $(N, 2 \cdot M)$ , where  $M$  is the number of spatial dimensions.

¥ **batch\_index** must be a tensor of shape  $(N, 1)$ .

¥ **size** must have an entry for each spatial dimension. The items in **size** must be positive.

##### Result semantics

Let the shape of **input** be  $(B, C, H, W)$ .

¥ The rank of **output** is the same as the rank of **input**. The shape of **output** is  $(N, C, h, w)$  where  $\text{size} = (h, w)$ .

The interpolation methods are the same as described in [Up and Down-Sampling](#).

Using operation `roi_resample`, it is possible to describe `roi_align` operations as compounds:

```
fragment avg_roi_align(  
    Ê input: tensor<scalar>,  
    Ê rois: tensor<scalar>,  
    Ê batch_index: tensor<integer>,  
    Ê output_size: integer[],  
    Ê sampling_rate: integer[],  
    Ê resize_method: string = 'symmetric' )  
-> ( output: tensor<scalar> )  
{  
    Ê size = [for i in range_of(output_size) yield output_size[i] * sampling_rate[i]];  
    Ê resized = roi_resample(input, rois, batch_index, output_size = size,  
    Ê                             method = resize_method);  
    Ê output = avg_pool(resized, size = sampling_rate, stride = sampling_rate);  
}
```

```
fragment max_roi_align(  
    Ê input: tensor<scalar>,  
    Ê rois: tensor<scalar>,  
    Ê batch_index: tensor<integer>,  
    Ê output_size: integer[],  
    Ê sampling_rate: integer[],  
    Ê resize_method: string = 'symmetric' )  
-> ( output: tensor<scalar> )  
{  
    Ê size = [for i in range_of(output_size) yield output_size[i] * sampling_rate[i]];  
    Ê resized = roi_resample(input, rois, batch_index, output_size = size,  
    Ê                             method = resize_method);  
    Ê output = max_pool(resized, size = sampling_rate, stride = sampling_rate);  
}
```

!

The literature contains a similar operation called 'RoI warp'. It can be considered a special case of RoI align, where the RoI coordinates are rounded to integer values before processing. This can be implemented using the `round` operation on the `rois` tensor before `roi_align` is called.

## 4.7. Matrix Multiplication

Matrix multiplication is a required operation for implementing fully connected linear layers in neural networks. `matmul` is a general matrix multiplication operation, which encompasses various special cases such as matrix-vector, vector-matrix and vector-vector operations.

```

fragment matmul (
  Ê A: tensor<scalar>,          # the left-hand-side argument
  Ê B: tensor<scalar>,          # the right-hand-side argument
  Ê transposeA: logical = false, # whether to transpose A
  Ê transposeB: logical = false ) # whether to transpose B
-> ( C: tensor<scalar> )        # the resulting tensor

```

### Argument validity

¥ The rank of **A** and **B** must be at least 2 and they must be equal. Let  $r = \text{rank}(A)$ . The first  $r - 2$  dimensions of **A** and **B** (if any) behave as batch dimensions, resulting in a batch of matrix multiplications (broadcasting may apply if needed, as for [Binary Operations](#)).

Let the shape of **A** be  $(b, m_A, n_A)$  and the shape of **B** be  $(b, m_B, n_B)$ . Then:

- ¥ If `transposeA = false` and `transposeB = false` then  $n_A$  must equal  $m_B$ .
- ¥ If `transposeA = false` and `transposeB = true` then  $n_A$  must equal  $n_B$ .
- ¥ If `transposeA = true` and `transposeB = false` then  $m_A$  must equal  $m_B$ .
- ¥ If `transposeA = true` and `transposeB = true` then  $m_A$  must equal  $n_B$ .

### Result semantics

- ¥ The rank of **C** will be  $r$ .
- ¥ If `transposeA = false` and `transposeB = false` then the shape of **C** will be  $(b, m_A, n_B)$ .
- ¥ If `transposeA = false` and `transposeB = true` then the shape of **C** will be  $(b, m_A, m_B)$ .
- ¥ If `transposeA = true` and `transposeB = false` then the shape of **C** will be  $(b, n_A, n_B)$ .
- ¥ If `transposeA = true` and `transposeB = true` then the shape of **C** will be  $(b, n_A, m_B)$ .

Let  $A' = A^T$  if `transposeA = true` and  $A' = A$  otherwise, and define  $B'$  similarly. Omitting optional batching for simplicity, let the shape of **C** be  $(m_C, n_C)$ , and let  $k$  be the agreeing dimension of the matrices  $A'$  and  $B'$ . Then, for all  $i \in [1..m_C]$  and  $j \in [1..n_C]$ :

$$C_{i,j} = \sum_{l=1}^k A'_{i,l} B'_{l,j}$$

## 4.8. Variable Updates

Variables are tensors that can be updated in each graph execution cycle, and the update is performed via a dedicated operation to avoid loops in the graph description and to have a well-defined execution order.

```

fragment update<?>(
  Ê variable: tensor<?>,          # the variable to be updated
  Ê value: tensor<?> )            # the new value
-> ( result: tensor<?> )          # the new value returned for convenience

```

## Argument validity

¥ The rank and shape of **value** must equal the shape of **variable**.

## Result semantics

¥ The rank and shape of **result** is equal to the shape of **variable**.

After executing all operations in the current graph cycle, the content of **variable** is replaced with the content of **value**. The variable itself always refers to the original value at the beginning of the cycle. The new value can be referenced via **result**. However, **result** is not a variable tensor, so it cannot be updated again. Thus, each variable can only be updated once in a cycle.

Note, that the computational graph that describes a single computation cycle in a potentially recurrent computation is itself acyclic.

# 4.9. Compound Operations

This section describes a set of compound fragments that are often used to build neural networks. The argument validity and result semantics of these fragments follow from the argument validity and result semantics of the primitives they are built from.



These definitions only provide the semantics for the operations, and do not prescribe the details of how these operations are to be implemented; they may be implemented as atomic operations or as a sequence of primitives that are mathematically equivalent to these definitions.

## 4.9.1. Activation Functions

```
fragment sigmoid( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
  y = 1.0 / (1.0 + exp(-x));
}
```

```
fragment relu( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
  y = max(x, 0.0);
}
```

```
fragment prelu( x: tensor<scalar>, alpha: tensor<scalar> ) -> ( y: tensor<scalar> )
{
  y = select(x < 0.0, alpha * x, x);
}
```

```

fragment leaky_relu( x: tensor<scalar>, alpha: scalar ) -> ( y: tensor<scalar> )
{
    y = prelu(x, alpha);
}

```

```

fragment elu( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = select(x < 0.0, exp(x) - 1.0, x);
}

```

```

fragment tanh( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = (exp(x) - exp(-x)) / (exp(x) + exp(-x));
}

```

```

fragment softmax( x: tensor<scalar>, axes: integer[] = [1] ) -> ( y: tensor<scalar> )
{
    m = max_reduce(x, axes = axes);
    e = exp(x - m);
    y = e / sum_reduce(e, axes = axes);
}

```

```

fragment softplus( x: tensor<scalar> ) -> ( y: tensor<scalar> )
{
    y = log(exp(x) + 1.0);
}

```

#### 4.9.2. Linear Operations

```

fragment linear(
    input: tensor<scalar>,
    filter: tensor<scalar>,
    bias: tensor<scalar> = 0.0 )
-> ( output: tensor<scalar> )
{
    output = matmul(input, filter, transposeB = true) + bias;
}

```

```

fragment separable_conv(
Ê   input: tensor<scalar>,
Ê   plane_filter: tensor<scalar>,
Ê   point_filter: tensor<scalar>,
Ê   bias: tensor<scalar> = 0.0,
Ê   border: string = 'constant',
Ê   padding: (integer,integer)[] = [],
Ê   stride: integer[] = [],
Ê   dilation: integer[] = [],
Ê   groups: integer = 1 )
-> ( output: tensor<scalar> )
{
Ê   filtered = conv(input, plane_filter, border = border, padding = padding,
Ê               stride = stride, dilation = dilation, groups = 0);
Ê   output = conv(filtered, point_filter, bias, groups = groups);
}

```

```

fragment separable_deconv(
Ê   input: tensor<scalar>,
Ê   plane_filter: tensor<scalar>,
Ê   point_filter: tensor<scalar>,
Ê   bias: tensor<scalar> = 0.0,
Ê   border: string = 'constant',
Ê   padding: (integer,integer)[] = [],
Ê   stride: integer[] = [],
Ê   dilation: integer[] = [],
Ê   output_shape: integer[] = [],
Ê   groups: integer = 1 )
-> ( output: tensor<scalar> )
{
Ê   filtered = deconv(input, point_filter, groups = groups);
Ê   output = deconv(filtered, plane_filter, bias, border = border,
Ê               padding = padding, stride = stride, dilation = dilation,
Ê               output_shape = output_shape, groups = 0);
}

```

### 4.9.3. Pooling Operations

```

fragment max_pool_with_index(
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer,integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [] )
-> ( output: tensor<scalar>, index: tensor<logical> )
{
  Ê index = argmax_pool(input, size = size, border = border, padding = padding,
  Ê                               stride = stride, dilation = dilation);
  Ê output = sample(input, index, size = size, border = border, padding = padding,
  Ê                               stride = stride, dilation = dilation);
}

```

```

fragment max_pool(
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer,integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [] )
-> ( output: tensor<scalar> )
{
  Ê output, index = max_pool_with_index(input, size = size, border = border,
  Ê                               padding = padding, stride = stride,
  Ê                               dilation = dilation);
}

```

```

fragment avg_pool(
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer,integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [] )
-> ( output: tensor<scalar> )
{
  Ê output = box(input, size = size, border = border, padding = padding,
  Ê                               stride = stride, dilation = dilation, normalize = true);
}

```

```

fragment rms_pool (
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê border: string = 'constant',
  Ê padding: (integer,integer)[] = [],
  Ê stride: integer[] = [],
  Ê dilation: integer[] = [] )
-> ( output: tensor<scalar> )
{
  Ê output = sqrt(avg_pool(sqr(input), size = size, border = border,
  Ê padding = padding, stride = stride, dilation = dilation));
}

```

#### 4.9.4. Normalization Operations

```

fragment local_response_normalization(
  Ê input: tensor<scalar>,
  Ê size: integer[],
  Ê alpha: scalar = 1.0,
  Ê beta: scalar = 0.5,
  Ê bias: scalar = 1.0 )
-> ( output: tensor<scalar> )
{
  Ê sigma = bias + alpha * box(sqr(input), size = size, normalize = true);
  Ê output = input / (sigma ^ beta);
}

```

```

fragment local_mean_normalization(
  Ê input: tensor<scalar>,
  Ê size: integer[] )
-> ( output: tensor<scalar> )
{
  Ê mean = box(input, size = size, normalize = true);
  Ê output = input - mean;
}

```



```

fragment local_variance_normalization(
    Ê input: tensor<scalar>,
    Ê size: integer[],
    Ê bias: scalar = 0.0,
    Ê epsilon: scalar = 0.0 )
-> ( output: tensor<scalar> )
{
    Ê sigma = box(sqr(input), size = size, normalize = true);
    Ê output = input / max(sqrt(sigma) + bias, epsilon);
}

```

```

fragment local_contrast_normalization(
    Ê input: tensor<scalar>,
    Ê size: integer[],
    Ê bias: scalar = 0.0,
    Ê epsilon: scalar = 0.0 )
-> ( output: tensor<scalar> )
{
    Ê centered = local_mean_normalization(input, size = size);
    Ê output = local_variance_normalization(centered, size = size, bias = bias,
    Ê                                     epsilon = epsilon);
}

```

```

fragment l1_normalization(
    Ê input: tensor<scalar>,
    Ê axes: integer[],
    Ê bias: scalar = 0.0,
    Ê epsilon: scalar = 0.0 )
-> ( output: tensor<scalar> )
{
    Ê sigma = sum_reduce(abs(input), axes = axes);
    Ê output = input / max(sigma + bias, epsilon);
}

```

```

fragment l2_normalization(
    Ê input: tensor<scalar>,
    Ê axes: integer[],
    Ê bias: scalar = 0.0,
    Ê epsilon: scalar = 0.0 )
-> ( output: tensor<scalar> )
{
    Ê sigma = sum_reduce(sqr(input), axes = axes);
    Ê output = input / max(sqrt(sigma) + bias, epsilon);
}

```

```

fragment batch_normalization(
    Ê input: tensor<scalar>,
    Ê mean: tensor<scalar>,
    Ê variance: tensor<scalar>,
    Ê offset: tensor<scalar>,
    Ê scale: tensor<scalar>,
    Ê epsilon: scalar )
-> ( output: tensor<scalar> )
{
    Ê output = offset + scale * (input - mean) / sqrt(variance + epsilon);
}

```

#### 4.9.5. Quantization Operations

```

fragment linear_quantize(
    Ê x: tensor<scalar>,
    Ê min: tensor<scalar>,
    Ê max: tensor<scalar>,
    Ê bits: integer )
-> ( y: tensor<scalar> )
{
    Ê r = scalar(2 ^ bits - 1);
    Ê z = clamp(x, min, max);
    Ê q = round((z - min) / (max - min) * r);
    Ê y = q / r * (max - min) + min;
}

```

```

fragment logarithmic_quantize(
    Ê x: tensor<scalar>,
    Ê max: tensor<scalar>,
    Ê bits: integer )
-> ( y: tensor<scalar> )
{
    Ê m = ceil(log2(max));
    Ê r = scalar(2 ^ bits - 1);
    Ê q = round(clamp(log2(abs(x)), m - r, m));
    Ê y = sign(x) * 2.0 ^ q;
}

```

#### 4.9.6. Miscellaneous Operations

```

fragment copy_n<?>( x: tensor<?>, times: integer ) -> ( y: tensor<?>[] )
{
    Ê y = [x] * times;
}

```

```
fragment add_n( x: tensor<scalar>[] ) -> ( y: tensor<scalar> )
{
    y = x[0] + add_n(x[1:]) if length_of(x) > 0 else 0.0;
}
```

```
fragment moments( input: tensor<scalar>, axes: integer[] )
-> ( mean: tensor<scalar>, variance: tensor<scalar> )
{
    mean = mean_reduce(input, axes = axes);
    variance = mean_reduce(sqr(input - mean), axes = axes);
}
```

# Chapter 5. Storing Network Data

The textual description introduced in the previous chapters is capable of capturing the neural network structure. However, to make descriptions of trained neural networks possible, the data parameters of the network (weight tensors) must be stored somehow, furthermore, they must be linked to the graph structure. Since variables, which form the data parameters of the network, are labelled uniquely, these labels can be used to link network data with the structure. The network weights are stored in separate data files that are named and stored into folders according to the labels of variables.

Further files may be used to store additional information attached to the basic structure, such as quantization information for tensors. For this reason, the files may be wrapped in a container to form a single data stream. This container may also provide compression and encryption mechanisms. The container format is not part of the NNEF specification, however, it is recommended to use the IEEE 1003.1-2008 tar archive with optional compression.

In what follows, the contents of the container and the tensor data files are described.

## 5.1. Container Organization

The contents of the container may consist of several files:

- ¥ A textual file describing the structure of the network, according to [Syntax](#). The file must be named 'graph.nnet'.
- ¥ A binary data file (structured according to [Tensor File Format](#)) for each variable tensor in the structure description, placed into sub-folders according to the labelling of the variables. The files must have '.dat' extension. For example, a variable with label 'conv1/filter' is placed into the folder 'conv1' under the name 'filter.dat'. Different versions of the same data (for example with different quantization) may be present starting with the same name, having additional (arbitrary) extensions after '.dat'.
- ¥ An optional quantization file (structured according to [Quantization File Format](#)) containing quantization algorithm details for [exported](#) tensors. The file must be named 'graph.quant'.

Note, that vendors may use the container to store vendor specific information in sub-folders and files, such as optimized network data in custom formats. This information is simply omitted by other tools.

## 5.2. Tensor File Format

Each tensor data file consists of two parts, the header and the data. All data is laid out in little-endian byte order. The header size is fixed to 128 bytes, and consists of the following (in this order):

- ¥ A two-byte magic number. The first byte is the ASCII character 'N' (0x4E) and the second byte is 0xEF.
- ¥ A two-byte version info (one byte for major, one for minor version).
- ¥ A 4-byte unsigned integer indicating the length of the actual tensor data in bytes (starting at

offset 128).

¥ A 4-byte unsigned integer indicating the rank of the tensor. Maximal value is 8.

¥  $8 \times$  4-byte unsigned integers denoting the extents of the tensor. Only relevant dimensions up to rank must be filled, the rest must be filled with zeros.

¥ A 4-byte unsigned integer indicating the number of bits per item, at most 64.

¥ A 4-byte code indicating the quantization (compression) algorithm used to store the data. The code is split to two 2-byte parts. The first part is a vendor code, the second part is an algorithm code. See reserved values below.

¥ 32 bytes for parameters of the quantization algorithm. The interpretation of these 32 bytes depends on the algorithm code.

The remaining bytes of the 128 byte header are reserved for future use and should be filled with zeros.

The quantization algorithm vendor code 0x0 identifies the Khronos Group. For the algorithm code part, Khronos specifies the following:

¥ 0x00 - uncompressed float values in IEEE format. Valid bits per item is 16, 32, 64.

¥ 0x01 - uncompressed integer values.

¥ 0x10 - quantized values with linear quantization as defined by the `linear_quantize` operation

¥ 0x11 - quantized values with logarithmic quantization as defined by the `logarithmic_quantize` operation

In case of uncompressed integer values (code 0x01), the first 4-byte value of the parameters signal whether the data-type is signed (zero indicates unsigned, non-zero indicates signed).

The codes 0x10 and 0x11 have corresponding parameters as follows:

¥ `min`: 32-bit IEEE floating point value indicating the value to which an item containing all-zero bits is mapped

¥ `max`: 32-bit IEEE floating point value indicating the value to which an item containing all-one bits is mapped

In case of the logarithmic quantization algorithm (0x11), the `min` parameter must be set to 0 (unsigned) or `-max` (signed).

Following the header, the tensor data is laid out linearly in a continuous row-major order, as a bit-stream packed continuously into a sequence of bytes. If the last group of bits does not fall into a byte boundary, the last byte is padded with zero bits.

The extents of the tensors must be equal to the ones calculated from the network description in 'graph.nnet'.

In the following the encoding and decoding formulas for quantized representations are detailed. Let  $b$  denote the number of bits per item, and let  $r = 2^b - 1$ . The quantized representation will contain integer values in  $[0..r]$ . Let  $x$  denote the input value, furthermore let  $[x]_a^b$  denote a value clamped to be within the range  $[a, b]$ .

Encoding with linear quantization is as follows:

$$q = \text{round}\left(\left[\frac{x - \min}{\max - \min}\right]_0^1 \cdot r\right)$$

Decoding with linear quantization is as follows:

$$\bar{x} = \frac{q}{r} \cdot (\max - \min) + \min$$

In case of logarithmic quantization, let  $m = \text{ceil}(\log_2 \max)$ , and suppose first that  $x \geq 0$ . Encoding with logarithmic quantization is as follows:

$$q = \text{round}([\log_2 x]_m^m - r) - (m - r)$$

Note that  $\log_2 0 = -\infty$  in the above equation.

Decoding with logarithmic quantization is as follows:

$$\bar{x} = 2^{q + (m - r)}.$$

If  $x$  is signed, then a sign bit must be allocated from the  $b$  bits available for the encoding, and the actual bit width to perform the above computation must be decreased by one. The computation must be performed on  $|x|$  and the sign information must be carried into the code  $q$ .

## 5.3. Quantization File Format

The quantization file has a simple line based textual format. Each line contains a tensor identifier (string in quotes, `' '` or `""`) and a quantization algorithm terminated by a semicolon. Identifiers must refer to [exported](#) activation tensor names (not variable labels) in the structure description:

```
...  
"filter1": linear_quantize(min = -2.0, max = 2.5, bits = 8);  
"bias1": linear_quantize(min = -3.0, max = 0.75, bits = 8);  
...
```

The quantization algorithm string must not contain an argument for the tensor to be quantized, but all other parameters must be provided as named arguments with compile-time expressions (no tensor ids). The tensor to be quantized is the one identified by the name before the separating colon (see [Quantization](#) for more info on interpretation of quantization info).

# Chapter 6. Document Validity

Validity and integrity of a document in the NNEF format can be checked independently of an implementation that executes it. Conceptually, a document processor should be able to take in a document, and if it is well-formatted, the processor should be able to output another document, which represents the same graph in a flattened format built only as a sequence of primitives with concrete arguments, without any compound fragment definitions and formal parameter expressions.

The process of validity checking consists of a number of steps described in detail in the previous chapters. The goal of this section is to summarize this process. As the result of this process, a document is either accepted as valid, or rejected as illegal. The cause of rejection may be various, depending on the stage in which an integrity problem is encountered during processing. The processing steps and the rejection causes are the following:

1. Syntactic parsing may result in syntax errors when the document does not conform to the grammar defined in [Syntax](#).
2. Semantic analysis may result in semantic errors when the syntactically valid document violates constraints defined in [Semantics](#). The violations may be of the following type:
  1. Invalid fragment definition, as defined in [Graph and Fragment Definition](#).
  2. Invalid argument structure in invocation, as defined in [Graph and Fragment Definition](#).
  3. Invalid type agreement, as defined in [Type System and Type Checking](#).
  4. Invalid formal identifier usage, as defined in [Identifier Usage in the Graph and Fragment Body](#).
3. The invocation of operations and the expansion of compound operations to primitive operations may result in invalid arguments in invocations. Validity of arguments is defined individually for each primitive in [Operations](#).
4. Loading serialized tensor data, as defined in [Storing Network Data](#), may result in conflicting tensor shapes (the stored shape conflicts with the shape in the structure definition).

When any of the above errors occur, the processor of the document should report the error and may stop processing. Tensor shape related argument validity checking (3.) and shape propagation may only be performed if no custom operations are defined in the document (for which shape propagation is not possible to define via the syntax). All other validity checks can still be performed in the presence of custom operations.

# Chapter 7. Quantization

Executing neural networks in quantized representations is an important aspect of high performance inference. To achieve best performance, neural networks need to be trained in special ways that takes the expected precision of the target inference hardware into account, therefore when transferring trained networks to inference engines, the quantization method used during training needs to be conveyed by NNEF. However, since NNEF aims to be a platform independent description of network structure and data, the description of quantization also needs to remain independent of the underlying representations of specific training frameworks and inference APIs.

To solve this seemingly contradictory issue, NNEF uses the concept of 'pseudo'-quantization. Note that NNEF deals with real valued tensors which are conceptually of infinite precision to avoid reference to machine representations. To keep this status, quantization operations convert tensor data into quantized values while conceptually remaining infinite precision real values. That is, arbitrary values are rounded and clamped to certain finite number of values depending on the exact algorithm. The quantization algorithms are described as compound operations (see [Quantization Operations](#)), therefore the set of quantization techniques can easily be extended.

Since different inference engines may have different hardware capabilities, quantization information is only meant as a hint for executing networks. It conveys the information that the network was trained with a specific precision and representation in mind, but the actual inference engine that executes the network may use a different representation or quantization level (which may or may not result in performance degradation).

## 7.1. Incorporating Quantization Info

Quantization can be applied to both network *activations* and stored tensor *parameters*. Furthermore, the tensor parameter data itself may be stored in a quantized or non-quantized format. Even if the parameters are not stored in a quantized format, they may be quantized during inference.

There are two ways to use the quantization operations in NNEF. First, since they are just like any other operation (input-output mapping, in this case taking a tensor and outputting its quantized version), they can simply be incorporated into a computational graph description.

The following example shows an excerpt from a possible quantized network where only activations are quantized:



```

input = tensor(shape = [1, 3, 224, 224]);
filter1 = variable(shape = [32, 3, 5, 5], label = 'conv1/filter');
bias1 = variable(shape = [1, 32], label = 'conv1/bias');
conv1 = relu(conv(input, filter1) + bias1);
quant1 = linear_quantize(conv1, min = 0.0, max = 1.0, bits = 8);
filter2 = variable(shape = [64, 32, 3, 3], label = 'conv2/filter');
bias2 = variable(shape = [1, 64], label = 'conv2/bias');
conv2 = relu(conv(quant1, filter2) + bias2);
quant2 = linear_quantize(conv2, min = 0.0, max = 1.0, bits = 8);
...

```

If we wanted to quantize parameters such as filters and biases as well by adding further quantization operations for them before the convolution operations, the description would soon become very cluttered. Furthermore, quantization operations are only hints to the inference engine and do not necessarily result in actual computation. To avoid cluttering the network structure, as a second option, quantization info may be placed into a separate file that relies on exported tensor [identifiers](#) to map tensors to quantization operations. We may keep the original core structure:

```

input = tensor(shape = [1, 3, 224, 224]);
filter1 = variable(shape = [32, 3, 5, 5], label = 'conv1/filter');
bias1 = variable(shape = [1, 32], label = 'conv1/bias');
conv1 = relu(conv(input, filter1) + bias1);
filter2 = variable(shape = [64, 32, 3, 3], label = 'conv2/filter');
bias2 = variable(shape = [1, 64], label = 'conv2/bias');
conv2 = relu(conv(quant1, filter2) + bias2);
...

```

and indicate quantization separately, which may be applied to external input parameters, constants, variables and activations as well:

```

"input": linear_quantize(min = 0.0, max = 1.0, bits = 8);
"filter1": linear_quantize(min = -1.0, max = 1.0, bits = 8);
"bias1": linear_quantize(min = -1.0, max = 1.0, bits = 8);
"conv1": linear_quantize(min = 0.0, max = 1.0, bits = 8);
"filter2": linear_quantize(min = -1.0, max = 1.0, bits = 8);
"bias2": linear_quantize(min = -1.0, max = 1.0, bits = 8);
"conv2": linear_quantize(min = 0.0, max = 1.0, bits = 8);
...

```

For the sake of explanation, linear quantization is used everywhere in this example, with ranges [0,1] for activations and [-1,1] for variables, but different parameters or algorithms could be used for each tensor. The two descriptions together are interpreted as if the appropriate quantization was applied to each tensor after its value is computed, which is equivalent to the following description:

```

input = linear_quantize(tensor(shape = [1, 3, 224, 224]),
    min = 0.0, max = 1.0, bits = 8);
filter1 = linear_quantize(variable(shape = [32, 3, 5, 5], label = 'conv1/filter'),
    min = -1.0, max = 1.0, bits = 8);
bias1 = linear_quantize(variable(shape = [1, 32], label = 'conv1/bias'),
    min = -1.0, max = 1.0, bits = 8);
conv1 = linear_quantize(relu(conv(input, filter1) + bias1),
    min = 0.0, max = 1.0, bits = 8);
filter2 = linear_quantize(variable(shape = [64, 32, 3, 3], label = 'conv2/filter'),
    min = -1.0, max = 1.0, bits = 8);
bias2 = linear_quantize(variable(shape = [1, 64], label = 'conv2/bias'),
    min = -1.0, max = 1.0, bits = 8);
conv2 = linear_quantize(relu(conv(conv1, filter2) + bias2),
    min = 0.0, max = 1.0, bits = 8);
...

```

In case when the parameter data are stored in a quantized format, only the quantization indices are stored, along with the quantization algorithm as a text string (see [Tensor File Format](#)). For example, when an algorithm quantizes to 8 bits, the minimum value is stored as index 0, the next value as index 1, and so on, until the maximum value which has index 255. In this case, a string such as `linear_quantize(min = -1.0, max = 1.0, bits = 8)` stored in the tensor file 'conv1/filter.dat' provides the interpretation for the 8-bit data. Hence the tensor `filter1` conceptually contains the real valued data as if it was pseudo-quantized (of course, and implementation need not explicitly perform this conversion). If the quantization file also contains a line for the key 'filter1', the quantization stored in the data file may be overridden for the purpose of actual computation. So, the quantization algorithm in the tensor file is for interpreting the data in that file, while the quantization algorithm in the quantization file is for the actual computation during execution. Quantization info in the quantization file may also be present for a variable if it is not stored in a quantized format in the data file.

## 7.2. Dynamic Quantization

The above examples showcase static quantization, where the quantization ranges are determined *a priori*, typically from the statistics of a test data set, and are fixed during inference. However, quantization may be dynamic as well, when the quantization range is calculated during inference. Note, that the definition of quantization operations let the quantization range be defined with tensors.

Dynamic quantization is only meaningful for activation tensors. In this case, the quantization operations must be part of the structural description, since the calculation of the range arguments must also be part of the computation. In the following example, the maximum of the range is calculated as the actual maximum of the data. Also, it is possible to calculate separate max values for each channel to perform channel-wise quantization:

```
...  
conv1 = relu(conv(input, filter1) + bias1);  
max1 = max_reduce(conv1, axes = [0, 2, 3]);  
quant1 = linear_quantize(conv1, min = 0.0, max = max1, bits = 8);  
...
```

Note, that even if the quantization operations are part of the description, they may be ignored by an implementation that does not have the capabilities to perform them (of course, at the cost of inaccuracies). In this case, the operations that calculate the range (`max_reduce` here) may also be ignored.

# Appendix A: Grammar Definitions

## A.1. Flat Grammar

```
<document> ::= <version> <extension>* <graph-definition>
<version> ::= "version" <numeric-literal> ";"
<extension> ::= "extension" <identifier>+ ";"

<graph-definition> ::= <graph-declaration> <body>
<graph-declaration> ::= "graph" <identifier> "(" <identifier-list> ")"
    È      "->" "(" <identifier-list> ")"
<identifier-list> ::= <identifier> ("," <identifier>)*

<body> ::= "{" <assignment>+ "}"
<assignment> ::= <lvalue-expr> "=" <invocation> ";"

<invocation> ::= <identifier> ["<" <type-name> ">"] "(" <argument-list> ")"
<argument-list> ::= <argument> ("," <argument>)*
<argument> ::= <rvalue-expr> | <identifier> "=" <rvalue-expr>

<lvalue-expr> ::= <identifier> | <array-lvalue-expr> | <tuple-lvalue-expr>
<array-lvalue-expr> ::= "[" [<lvalue-expr> ("," <lvalue-expr>)* ] "]"
<tuple-lvalue-expr> ::= "(" <lvalue-expr> ("," <lvalue-expr>)+ ")" |
    È      <lvalue-expr> ("," <lvalue-expr>)+

<rvalue-expr> ::= <identifier> | <literal> | <array-rvalue-expr> | <tuple-rvalue-expr>
<array-rvalue-expr> ::= "[" [<rvalue-expr> ("," <rvalue-expr>)* ] "]"
<tuple-rvalue-expr> ::= "(" <rvalue-expr> ("," <rvalue-expr>)+ ")"

<literal> ::= <numeric-literal> | <string-literal> | <logical-literal>
```

## A.2. Compositional Grammar

```
<document> ::= <version> <extension>* <fragment-definition>* <graph-definition>
<version> ::= "version" <numeric-literal> ";"
<extension> ::= "extension" <identifier>+ ";"

<fragment-definition> ::= <fragment-declaration> (<body> | ";" )
<fragment-declaration> ::= "fragment" <identifier> [<generic-declaration>]
    È      "(" <parameter-list> ")" "->" "(" <result-list> ")"
<generic-declaration> ::= "<" "?" ["=" <type-name> ] ">"
<parameter-list> ::= <parameter> ("," <parameter>)*
<parameter> ::= <identifier> ":" <type-spec> ["=" <literal-expr>]
<result-list> ::= <result> ("," <result>)*
<result> ::= <identifier> ":" <type-spec>

<literal-expr> ::= <literal> | <array-literal-expr> | <tuple-literal-expr>
```

```

<literal> ::= <numeric-literal> | <string-literal> | <logical-literal>
<array-literal-expr> ::= "[" [<literal-expr> ("," <literal-expr>)* ] "]"
<tuple-literal-expr> ::= "(" <literal-expr> ("," <literal-expr>)+ ")"

<type-spec> ::= <type-name> | <tensor-type-spec> |
Ê          <array-type-spec> | <tuple-type-spec>
<type-name> ::= "integer" | "scalar" | "logical" | "string" | "?"
<tensor-type-spec> ::= "tensor" "<" [<type-name>] ">"
<array-type-spec> ::= <type-spec> "[]"
<tuple-type-spec> ::= "(" <type-spec> ("," <type-spec>)+ ")"

<graph-definition> ::= <graph-declaration> <body>
<graph-declaration> ::= "graph" <identifier> "(" <identifier-list> ")"
Ê          "->" "(" <identifier-list> ")"
<identifier-list> ::= <identifier> ("," <identifier>)*

<body> ::= "{" <assignment>+ "}"
<assignment> ::= <lvalue-expr> "=" <rvalue-expr> ";"

<lvalue-expr> ::= <identifier> | <array-lvalue-expr> | <tuple-lvalue-expr>
<array-lvalue-expr> ::= "[" [<lvalue-expr> ("," <lvalue-expr>)* ] "]"
<tuple-lvalue-expr> ::= "(" <lvalue-expr> ("," <lvalue-expr>)+ ")" |
Ê          <lvalue-expr> ("," <lvalue-expr>)+

<invocation> ::= <identifier> "[" "<" <type-name> ">" ] "(" <argument-list> ")"
<argument-list> ::= <argument> ("," <argument>)*
<argument> ::= <rvalue-expr> | <identifier> "=" <rvalue-expr>

<rvalue-expr> ::= <identifier>
Ê          | <literal>
Ê          | <binary-expr>
Ê          | <unary-expr>
Ê          | <paren-expr>
Ê          | <array-rvalue-expr>
Ê          | <tuple-rvalue-expr>
Ê          | <subscript-expr>
Ê          | <if-else-expr>
Ê          | <comprehension-expr>
Ê          | <builtin-expr>
Ê          | <invocation>

<array-rvalue-expr> ::= "[" [<rvalue-expr> ("," <rvalue-expr>)* ] "]"
<tuple-rvalue-expr> ::= "(" <rvalue-expr> ("," <rvalue-expr>)+ ")"

<unary-expr> ::= <unary-operator> <rvalue-expr>
<binary-expr> ::= <rvalue-expr> <binary-operator> <rvalue-expr>
<paren-expr> ::= "(" <rvalue-expr> ")"

<binary-operator> ::= <comparison-operator>
Ê          | <binary-arithmetic-operator>
Ê          | <binary-logical-operator>

```

```

<compari son-operator> ::= "<" | "<=" | ">" | ">=" | "==" | "!=" | "in"
<bi nary-ari thmeti c-operator> ::= "+" | "-" | "*" | "/" | "^"
<bi nary-l ogi cal -operator> ::= "&&" | "||"

<unary-operator> ::= <unary-ari thmeti c-operator>
Ê | <unary-l ogi cal -operator>
<unary-ari thmeti c-operator> ::= "+" | "-"
<unary-l ogi cal -operator> ::= "!"

<i f-el se-expr> ::= <rval ue-expr> "i f" <rval ue-expr> "el se" <rval ue-expr>

<l oop-i ter> ::= <i denti fi er> "i n" <rval ue-expr>
<l oop-i ter-l i st> ::= <l oop-i ter> ("," <l oop-i ter>)*
<comprehen si on-expr> ::= "[" "for" <l oop-i ter-l i st> ["i f" <rval ue-expr>]
Ê "yi el d" <rval ue-expr> "]"

<subscri pt-expr> ::= <rval ue-expr> "[" (<rval ue-expr> |
Ê [<rval ue-expr>] ":" [<rval ue-expr>]) "]"

<bui l ti n-expr> ::= <bui l ti n-name> "(" <rval ue-expr> ")"
<bui l ti n-name> ::= "shape_of" | "length_of" | "range_of"
Ê | "integer" | "scal ar" | "logi cal " | "string"

```

# Appendix B: Example Export

## B.1. AlexNet

The following example shows a complete description of AlexNet as exported from TensorFlow. Note, that the example uses only flat NNEF syntax.

```

graph AlexNet( input ) -> ( output )
{
  Ê input = external(shape = [1, 3, 224, 224]);
  Ê kernel1 = variable(shape = [64, 3, 11, 11], label = 'alexnet_v2/conv1/kernel');
  Ê bias1 = variable(shape = [1, 64], label = 'alexnet_v2/conv1/bias');
  Ê conv1 = conv(input, kernel1, bias1, padding = [(0,0), (0,0)],
  Ê           border = 'constant', stride = [4, 4], dilation = [1, 1]);
  Ê relu1 = relu(conv1);
  Ê pool1 = max_pool(relu1, size = [1, 1, 3, 3], stride = [1, 1, 2, 2],
  Ê          border = 'ignore', padding = [(0,0), (0,0), (0,0), (0,0)]);
  Ê kernel2 = variable(shape = [192, 64, 5, 5], label = 'alexnet_v2/conv2/kernel');
  Ê bias2 = variable(shape = [1, 192], label = 'alexnet_v2/conv2/bias');
  Ê conv2 = conv(pool1, kernel2, bias2, padding = [(2,2), (2,2)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê relu2 = relu(conv2);
  Ê pool2 = max_pool(relu2, size = [1, 1, 3, 3], stride = [1, 1, 2, 2],
  Ê          border = 'ignore', padding = [(0,0), (0,0), (0,0), (0,0)]);
  Ê kernel3 = variable(shape = [384, 192, 3, 3], label = 'alexnet_v2/conv3/kernel');
  Ê bias3 = variable(shape = [1, 384], label = 'alexnet_v2/conv3/bias');
  Ê conv3 = conv(pool2, kernel3, bias3, padding = [(1,1), (1,1)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê relu3 = relu(conv3);
  Ê kernel4 = variable(shape = [384, 384, 3, 3], label = 'alexnet_v2/conv4/kernel');
  Ê bias4 = variable(shape = [1, 384], label = 'alexnet_v2/conv4/bias');
  Ê conv4 = conv(relu3, kernel4, bias4, padding = [(1,1), (1,1)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê relu4 = relu(conv4);
  Ê kernel5 = variable(shape = [256, 384, 3, 3], label = 'alexnet_v2/conv5/kernel');
  Ê bias5 = variable(shape = [1, 256], label = 'alexnet_v2/conv5/bias');
  Ê conv5 = conv(relu4, kernel5, bias5, padding = [(1,1), (1,1)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê relu5 = relu(conv5);
  Ê pool3 = max_pool(relu5, size = [1, 1, 3, 3], stride = [1, 1, 2, 2],
  Ê          border = 'ignore', padding = [(0,0), (0,0), (0,0), (0,0)]);
  Ê kernel6 = variable(shape = [4096, 256, 5, 5], label = 'alexnet_v2/fc6/kernel');
  Ê bias6 = variable(shape = [1, 4096], label = 'alexnet_v2/fc6/bias');
  Ê conv6 = conv(pool3, kernel6, bias6, padding = [(0,0), (0,0)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê relu6 = relu(conv6);
  Ê kernel7 = variable(shape = [4096, 4096, 1, 1], label = 'alexnet_v2/fc7/kernel');
  Ê bias7 = variable(shape = [1, 4096], label = 'alexnet_v2/fc7/bias');
  Ê conv7 = conv(relu6, kernel7, bias7, padding = [(0,0), (0,0)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê relu7 = relu(conv7);
  Ê kernel8 = variable(shape = [1000, 4096, 1, 1], label = 'alexnet_v2/fc8/kernel');
  Ê bias8 = variable(shape = [1, 1000], label = 'alexnet_v2/fc8/bias');
  Ê conv8 = conv(relu7, kernel8, bias8, padding = [(0,0), (0,0)],
  Ê          border = 'constant', stride = [1, 1], dilation = [1, 1]);
  Ê output = softmax(conv8);
}

```



# Appendix C: Credits

NNEF 1.0 is the result of contributions from many people and companies participating in the Khronos NNEF Working Group, as well as input from the NNEF Advisory Panel. Members of the Working Group, including the company that they represented at the time of their contributions, are listed below. Some specific contributions made by individuals are listed together with their name.

- ¥ Peter McGuinness, independent (Working Group Chair)
- ¥ Frank Brill, Cadence
- ¥ Maciej Urbanski, Intel
- ¥ Ofer Rosenberg, Qualcomm
- ¥ Radha Giduthuri, AMD
- ¥ Sandip Parikh, Cadence
- ¥ Viktor Gyenes, Alimotive (Specification Editor, original format proposal)
- ¥ Xin Wang, Verisilicon

In addition to the Working Group, the NNEF Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

Administrative support to the Working Group was provided by members of Khronos Group, including Kathleen Mattson. Technical support was provided by James Riordon, webmaster of Khronos.org.

# Appendix D: List of Changes

Version 1.0 (since Provisional):

- ¥ renamed `extent` type to `integer`
- ¥ made semicolon after assignments mandatory
- ¥ depth-wise convolution can be expressed with `groups = 0`
- ¥ added new operations: `squeeze`, `unsqueeze`, `stack`, `unstack`, `slice`, `argmax_reduce`, `prelu`, `RoI` operations
- ¥ `matmul` operation generalized to batched version, `trA` and `trB` parameters renamed to `transposeA` and `transposeB`
- ¥ renamed `perm` parameter of `transpose` operation to `axes`
- ¥ added `epsilon` parameter to normalization operations
- ¥ variable labels are restricted to contain limited set of characters allowed in file names, case insensitive comparison
- ¥ added missing negative numeric literals in case of flat syntax
- ¥ changed syntax of array comprehension to let the loop variable be declared before the loop core expression
- ¥ added the `in` operator for testing containment of items in arrays
- ¥ made parentheses mandatory for tuple literals in rvalue expressions
- ¥ added appendix that contains both grammars in one place
- ¥ added syntax to specify extensions in the structure description
- ¥ revised tensor binary header (fixed size, aligned fields, quantization info is represented with binary fields instead of text)
- ¥ introduced syntax for explicit tensor data-type specification for operations
- ¥ introduced explicit generic tensor data-type syntax
- ¥ clarifications about type system and casting
- ¥ clarified tensor rank definition, added explicit tensor rank pre and post conditions for operations
- ¥ added `output_shape` parameter to `deconv`-like operations
- ¥ removed the option of using 0s in the `external` operation to indicate unknown shapes
- ¥ enhancement of formulas and wording in various places

Version 1.0.1:

- ¥ deprecation of `shape_of` builtin function
- ¥ deprecation of the use of keyword arguments for tensor arguments in operation invocations (only positional)
- ¥ prohibiting the use of `variable` and `update` operation inside fragments

- ¥ prohibiting the declaration of fragments that do not have tensor results
- ¥ restriction of the left-hand-side of assignments to be structurally equivalent to the result type of invoked operations (to let result array sizes be well defined)
- ¥ allow output tensors to be assigned a constant value inside fragments
- ¥ removed appendix of layer and network examples (depended on `shape_of` function)
- ¥ replaced `stack`, `unstack`, `squeeze`, `unsqueeze` compound operation definitions with primitives
- ¥ removed appendix that listed the operation mappings (content moved to GitHub)
- ¥ added change-log appendix
- ¥ typo fixes and wording clarifications