Architecture Specification Language

Language Reference Manual

Document number	DDI0612
Document quality	BET
Document version	00bet7
Document confidentiality	Non-confidential

Architecture Specification Language

Proprietary Notice

Arm document reference: 13203 - Version 1 (February 2022)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is at Beta quality. All major features of the ASL language are described in the manual, but some details might be subject to change. At this quality level the release will be sufficiently stable for initial product development.

Web Address

Arm welcomes feedback on this document. To give feedback on this document, please contact the ASL Steering Board, using the `asl-support@arm.com` alias, or other contact details that you may have received from Arm.

The latest version of this document can be found at `https://developer.arm.com/documentation/DDI0612`.

Contents

Architecture Specification Language

Preface

About this book

Abstract

The increasing importance of the Arm architecture coupled with the increasing use of formal verification of both hardware and software make it important to have a readable, precise way of describing the majority of the Arm architecture. This specification defines Arm's *Architecture Specification Language* (ASL) which is the language used in Arm's architecture reference manuals to describe the Arm architecture.

Structure of this document

This document has the following structure.

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for source code examples.

Also used in the main text for references to other items appearing in source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://developer.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`.

Rules-based writing

This specification consists of a set of individual Declarations and Rules.

An implementation which is compliant with this specification must conform to all of the Declarations and Rules in this specification.

Declarations and Rules must not be read in isolation, and where more than one exists relating to a particular feature, they are grouped into sections and subsections to provide the proper context. Where appropriate, these sections contain a short introduction to aid the reader.

Declaration

Each Declaration is identified by the letter D.

A Declaration is a statement which either:

- introduces the meaning of a concept or term, or
- describes the structure or encoding of data.

A Declaration does not describe behaviour.

Goal

A Goal is a statement about the purpose of a set of rules.

A Goal explains why a particular feature has been included in the specification.

A Goal is comparable to a “business requirement” or an “emergent property.”

A Goal is intended to be upheld by the logical conjunction of a set of rules.

A Goal is rendered with the label *G*.

Rule

Each Rule is identified by the letter R.

A Rule is a statement which either:

- describes the behaviour of part of ASL, or
- describes a requirement which must be met by specifications written in ASL.

A Rule does not define concepts or terminology.

Rationale

Some Declarations and Rules are accompanied by Rationale statements which explain why ASL was specified as it was. Rationale statements are identified by the letter X.

Information

Some sections contain additional information and guidance. This information and guidance is provided purely as an aid to understanding this specification. Information statements are identified by the letter I.

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

Declarations, Rules, Rationale statements and Information statements are collectively referred to as *content items*.

Identifiers

Each content item may have an associated identifier which is unique within the context of this specification.

When the document is prior to beta status:

- Content items are assigned numerical identifiers, in ascending order through the document (*0001*, *0002*, . . .).
- Identifiers are volatile: the identifier for a given content item may change between versions of the document.

After the document reaches beta status:

- Content items are assigned random alphabetical identifiers (*HJQS*, *PZWL*, . . .).
- Identifiers are preserved: a given content item has the same identifier across versions of the document.

Examples

Below are examples showing the appearance of each type of content item.

D This is a declaration statement.

G This is a goal statement.

R This is a rule statement.

R$_{\texttt{X001}}$ This is a rule statement with the unique identifier `X001`.

I This is an information statement.

X This is a rationale statement.

Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer http://developer.arm.com for access to Arm documentation.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title (Architecture Specification Language).
- The number (DDI0612 00bet7).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Inclusive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change. To report offensive language in this document, email `terms@arm.com`.

Open issues

Issues expected to be addressed in ASL Release 1.0

- Documentation of semantics of the language (in a mathematical style) is available in a separate document. This document is at an earlier stage of completeness.

Issues likely to be addressed in a future ASL specification

At this time we can only give an approximate indication of future directions, but some areas are being investigated for improvement:

- Language features to support composing larger specifications out of smaller components. These features might support the specification of modules and interfaces.
- Features to allow specification of concurrent execution.
- Features to allow for architectural unknown values, known as Z-Types in previous releases of this document.

Language versioning

Each iteration of the language has a version number in the form `X.Y.Z` where `X`, `Y`, and `Z` are non-negative integers.

Symbol	Meaning
`X`	is the major version
`Y`	is the minor version
`Z`	is the document version

Major version

Major version `X` is incremented when backwards incompatible changes are introduced. This may also include minor and document level changes. Document and minor version are reset to `0` when the major version is incremented.

Minor version

Minor version `Y` is incremented when new, backwards compatible functionality is introduced. It is incremented if any functionality is marked as deprecated. This may also include document level changes. Document version is reset to `0` when the minor version is incremented.

Document version

The document version `Z` is incremented when clarifications and corrections are introduced.

Pre-release version

A pre-release version is denoted by a hyphen and a predefined label: `dev`, `alpha`, `beta` or `rel`.

Example: `1.0.0-alpha`, `1.0.0-beta`

Revision

The revision is denoted by a hyphen and a positive integer immediately after the pre-release version.

Example: `1.0.0-alpha-1`, `1.0.0-rel-2`

Precedence

Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor and document versions are always compared numerically.

Example: `1.0.0` < `2.0.0` < `2.1.0` < `2.1.1`

Example: `1.2.0` < `1.19.0`

Example: `1.1.2` < `1.1.19`

When major, minor, and document are equal, a pre-release version has lower precedence than a normal version.

Example: `1.0.0-alpha` < `1.0.0`

Precedence for two pre-release versions with the same major, minor, and document version is determined by looking at the pre-release version and revision. For the pre-release version the precedence is `dev` < `alpha` < `beta` <

`rel`. For the revision the precedence is determined numerically. A matching version with a revision has a higher precedence than one without.

Example: `1.0.0-alpha` < `1.0.0-alpha-1` < `1.0.0-beta` < `1.0.0-beta-2` < `1.0.0-beta-11` < `1.0.0-rel-1` < `1.0.0`

Chapter 1
Introduction

The *Architecture Specification Language* (ASL) is designed and used to specify architectures. As a formal specification language it is designed to be accessible, understandable and unambiguous to programmers, hardware engineers and hardware verification engineers who collectively have quite a small intersection of languages they all understand. It can intentionally under specify behaviors in the architecture being described.

ASL is:

- a first-order language with strong static type-checking.
- whitespace-insensitive.
- imperative.

ASL has support for:

- bitvectors:
  - as a type.
  - as a literal constant.
  - bitvector concatenation.
  - bitvector constants with wildcards.
  - bitslices.
  - dependent types to support function overloading using bitvector lengths.
  - dependent types to reason about lengths of bitvectors.
- unbounded arithmetic types "*integer*" and "*real*".
- type inference.
- exceptions.
- enumerations.
- arrays.
- records.

ASL does not have support for:

- references or pointers.

A *specification* consists of a self-contained collection of ASL code.

1.1 Specifications

D_{GVBK} A *specification* is the set of *declarations* written in ASL code which describe an architecture.

I_{NXKD} See Chapter 4 *Declaration syntax* for details about ASL declarations.

1.1.1 Example specification 1

This is a very small example of a specification written in ASL. It consists of the following declarations:

- Global bitvectors `R0`, `R1` and `R2` representing the state of the system.
- A function `MyOR` demonstrating a simple bit-wise OR function of 2 bitvectors.
- Initialization of `R0` and `R1` bitvectors.
- Assignment of bitvector `R2` with the result of a function call.

Listing 1.1: Example specification 1

```
var R0: bits(4) = '0001';
var R1: bits(4) = '0010';
var R2: bits(4);

func MyOR{M}(x: bits(M), y: bits(M)) => bits(M)
begin
    return x OR y;
end

func reset()
begin
    R2 = MyOR(R0, R1);
end
```

1.1.2 Example specification 2

This is a very small example of a specification written in ASL. It consists of the following declarations:

- A global variable `COUNT` representing the state of the system.
- A procedure `ColdReset` to initialize the state of the system when power is applied and the system is reset. This interpretation of the function is a convention used in this particular specification. It is up to each specification to decide the role of each function.
- A procedure `Step` to advance the state of the system. That is, it defines the *transition relation* of the system. Again, this interpretation is a convention used in this particular specification, not part of the ASL language itself.

Listing 1.2: Example specification 2

```
var COUNT: integer;

func ColdReset()
begin
    COUNT = 0;
end

func Step()
begin
    assert COUNT >= 0;
    COUNT = COUNT + 1;
    assert COUNT > 0;
end
```

1.1.3 Example specification 3

This is a very small example of a specification in ASL. It consists of the following declarations:

- A function `Dot8` which operates on 2 bitvectors a byte at a time.

- A global variable `COUNT` to indicate the number of calls to the `Fib` function.
- A function `Fib` demonstrating recursion.
- Assignment of a global bitvector `X` with a call to the `Dot8` function.
- Assignment of a variable from the result of a call to the recursive function `Fib`.
- A function `main` which complies with the requirements of Chapter 10 *Runtime Environment*.

Listing 1.3: Example specification 3

```
func Dot8{N}(a: bits(N), b: bits(N)) => bits(N)
begin
    var n: integer = 0;
    for i = 0 to (N DIV 8) do
        n = n + UInt(a[(i * 8) +: 8]) * UInt(b[(i * 8) +: 8]);
    end
    return n[0 +: N];
end

var X: bits(16) = '1010 1111 0101 0000';

var COUNT: integer = 0;

func Fib(n: integer) => integer
begin
    COUNT = COUNT + 1;
    if n < 2 then
        return 1;
    else
        return Fib(n - 1) + Fib(n - 2);
    end
end

func main() => integer
begin
    X = Dot8(X, X);
    var fib10 = Fib(10);
    return 0;
end
```

1.1.4 Specification errors

A number of features of ASL can result in errors either when parsing or type-checking the specification or in execution or reasoning about the specification. A non-exhaustive list of examples include:

- assertion failure (see 6.6 *Assertion statements*)
- array index out of bounds (see 5.9 *Arrays, bitslices and invoking getter functions*)
- bitslice index out of bounds or bitslice width negative (see 5.9.1 *Bitslices*)
- no matching term in a case statement (see 6.8 *Case Statements*)
- uncaught specification exceptions (see 6.10 *Exception handling*)
- unbounded loops
- unbounded recursion
- division by zero (see 8.6.3 *Integer operations*)

These and other examples are discussed in more detail in the rest of this manual. The existence of an error in the specification indicates that the specification contains a bug and it is necessary to consult the specification author to obtain a corrected specification in order to use the specification.

Chapter 2
Lexical structure

This section describes the lexical structure of ASL.

2.1 Notational conventions

Tokens in the grammar of ASL have rule names which are enclosed in “<” and “>”. The enclosing “<” and “>” are omitted in Backus-Naur Form (BNF) productions which make use of the tokens.

Token rules are not described using BNF productions. Instead they are described using a simple Regular Expression syntax whose matching rules are as follows:

Table 2.1: lexical regular expressions

RegExp	Matches
`'c'`	the character `c`
`( A )`	A
`A B`	A followed by B
`A | B`	A or B
`A - B`	A but not B
`A *`	zero or more repetitions of A
`A +`	one or more repetitions of A
`["a_string"]`	Any character in a_string
`{"a_string"}`	The string a_string

2.2 Tokens

R_{GFSH} Files contain printable ASCII characters, carriage return and line feed.

I$_{GFJP}$ Printable ASCII characters have the decimal encoding of 32 through 126 (inclusive). This includes the space character (dec 32).

2.2.1 Comments and other whitespace

Listing 2.1: Definition of a comment

```
<comment> ::= {"//"} { <any except newline> }
            | {"/*"} { <any including newline> } {"*/"}
```

I$_{XFKN}$ Comments do not nest and the two styles of comments do not interact with each other.

```
    // Comment example 1.
    // In next line, the two strings "/*" and "*/" are regular characters within the comment
    // start of comment, /* still in comment */ and still in comment which ends with newline

    /* line 1 of example 2, a single comment 4 lines long.
       line 2 of the comment
    // line 3 of the comment, the "//" at start of this line are just regular characters
    // line 4 of the comment, this 4 line comment ends with these two characters -->*/

    /* L1 Comment example 3, shows you cannot nest or mix comment styles.
    /* L2 Note the declaration of the storage FOO on L6, is outside of the comment.
    /* L3 Note the first two characters on L6 do NOT start a nested comment.
    /* L4 However, the two chars '*' and '/' following the line number L6, terminate
    /* L5 the comment started on L1.
    /* L6 */ var FOO : integer = 1; // The declaration of FOO is not within any comment */
    /* L7 The last two characters on line L6 have no special meaning, */
    /* L8  they are just characters within the comment that started with the "//". */
```

R$_{SWQD}$ Comments, newlines and space characters are treated as whitespace.

R$_{FRWD}$ It is an error to use a tab character in ASL files.

2.2.2 Literals

Listing 2.2: Definition of a literal

```
literal_expr ::= int_lit
               | hex_lit
               | real_lit
               | bitvector_lit
               | string_lit
               | boolean_lit
```

R$_{HYFH}$ Integers are written either in decimal using one or more of the characters `0-9` and underscore, or in hexadecimal using `0x` at the start followed by the characters `0-9`, `a-f`, `A-F` and underscore. An integer literal cannot start with an underscore.

Listing 2.3: Definition of an integer

```
<int_lit> ::= digit ('_' | digit)*
```

Listing 2.4: Definition of a hexadecimal

```
<hex_lit> ::= '0' 'x' (digit | ["abcdefABCDEF"]) ('_' | digit | ["abcdefABCDEF"])*
```

R$_{QQBB}$ Fixed point real numbers are written in decimal and consist of one or more decimal digits, a decimal point and one or more decimal digits. Underscores can be added between digits to aid readability.

Listing 2.5: Definition of a real

```
<real_lit> ::= digit ('_' | digit)* '.' digit ('_' | digit)*
```

IHJCD Underscores in numbers are not significant, and their only purpose is to separate groups of digits to make constants such as `0xefff_fffe`, `1_000_000` or `3.141_592_654` easier to read.

R MXPS Boolean literals are written using `TRUE` or `FALSE`.

Listing 2.6: Definition of a boolean

```
<boolean_lit> ::= ( {"TRUE"} | {"FALSE"} )
```

R PBFK Constant bit-vectors are written using 1, 0 and spaces surrounded by single-quotes.

Listing 2.7: Definition of a bitvector

```
<bitvector_lit> ::= '\'' ["01 "]* '\''
```

I QCZX The spaces in a bitvector are not significant and are only used to improve readability. For example `'1111 1111 ↪1111 1111'` is the same as `'1111111111111111'`.

R RYMD Constant bit-masks are written using `1`, `0`, `x` and spaces surrounded by single-quotes. The `x` represents a don't care character.

Listing 2.8: Definition of a mask

```
<bitmask_lit> ::= '\'' ["01x "]* '\''
```

I PBPQ The spaces in a constant bit-mask are not significant and are only used to improve readability.

R ZRVY A string value is a string of zero or more characters, where a character is a printable ASCII character, tab (ASCII code 0x09) or newline (ASCII code 0x0A). String values are created by string literals.

String literals consist of printable characters surrounded by double-quotes. Actual tabs and newlines are not permitted in string literals, meaning that string literals cannot span multiple source lines. The backslash character, '\', is treated as an escape character.

Listing 2.9: Definition of a string

```
<string_lit> ::= '"' ((char - ["\"\\"]) | ('\\' ["nt\"\\"] ) )* '"'
```

Escape sequence	Meaning
`\n`	The newline, ASCII code `0x0A`
`\t`	The tab, ASCII code `0x09`
`\\`	The backslash character, `\`
`\"`	The double-quote character, `"`

I ZQSD An enumeration literal is also classed as a literal constant, but is syntactically an identifier.

2.2.3 Identifiers

R HPRD Identifiers start with a letter or underscore and continue with zero or more letters, underscores or digits.

I VQBX Identifiers are case sensitive. To improve readability, it is recommended to avoid the use of identifiers that differ only by the case of some characters.

R QMDM Reserved identifiers and the elements of `boolean_lit` cannot be used as identifiers.

R_{JGRK} The `reserved_id` grammar rule defines identifiers which are currently not permitted in ASL.

I_{SCLY} The `reserved_id` grammar rule includes keywords that are reserved for future use.

I_{HVLX} By convention, identifiers that begin with double-underscore are reserved for use in the implementation and should not be used in specifications.

Listing 2.10: Definition of a identifier

```
<identifier> ::= ( letter | '_' ) ( letter | '_' | digit )*
```

Listing 2.11: Reserved identifiers

```
<reserved_id> ::= "AND"          | "DIV"          | "DIVRM"        | "EOR"
                | "IN"           | "MOD"          | "NOT"          | "OR"
                | "SAMPLE"       | "UNKNOWN"      | "UNSTABLE"     | "XOR"
                | "_"            | "access"       | "advice"       | "after"
                | "any"          | "array"        | "as"           | "aspect"
                | "assert"       | "assume"       | "assumes"      | "before"
                | "begin"        | "bit"          | "bits"         | "boolean"
                | "call"         | "case"         | "cast"         | "catch"
                | "class"        | "config"       | "constant"     | "dict"
                | "do"           | "downto"       | "else"         | "elsif"
                | "end"          | "endcase"      | "endcatch"     | "endclass"
                | "endevent"     | "endfor"       | "endfunc"      | "endgetter"
                | "endif"        | "endmodule"    | "endnamespace" | "endpackage"
                | "endproperty"  | "endrule"      | "endsetter"    | "endtemplate"
                | "endtry"       | "endwhile"     | "entry"        | "enumeration"
                | "event"        | "exception"    | "export"       | "expression"
                | "extends"      | "extern"       | "feature"      | "for"
                | "func"         | "get"          | "getter"       | "gives"
                | "if"           | "iff"          | "implies"      | "import"
                | "in"           | "integer"      | "intersect"    | "intrinsic"
                | "invariant"    | "is"           | "let"          | "list"
                | "map"          | "module"       | "namespace"    | "newevent"
                | "newmap"       | "of"           | "original"     | "otherwise"
                | "package"      | "parallel"     | "pass"         | "pattern"
                | "pointcut"     | "port"         | "pragma"       | "private"
                | "profile"      | "property"     | "protected"    | "public"
                | "real"         | "record"       | "repeat"       | "replace"
                | "requires"     | "rethrow"      | "return"       | "rule"
                | "set"          | "setter"       | "shared"       | "signal"
                | "statements"   | "string"       | "subtypes"     | "template"
                | "then"         | "throw"        | "to"           | "try"
                | "type"         | "typeof"       | "union"        | "until"
                | "using"        | "var"          | "watch"        | "when"
                | "where"        | "while"        | "with"         | "ztype"
```

2.2.4 Delimiters

Listing 2.12: Definition of a delimiter

```
<delimiter> ::= "!"   | "!="  | "&&"  | "("   | ")"   | "*"   | "*:"  | "+"   | "+:"
              | ","   | "-"   | "-->" | "."   | ".."  | "/"   | ":"   | ";"   | "<"
              | "<->" | "<<"  | "<="  | "="   | "=="  | "=>"  | ">"   | ">="  | ">>"
              | "@"   | "["   | "]"   | "^"   | "{"   | "||"  | "}"
```

2.3 Pragmas

I_{SNBZ} Pragmas are allowed to modify the lexical structure when parsing an ASL specification.

I_{LRZB} See 4.6 *Pragmas* and 6.11 *Pragmas* for the definition of pragmas.

I_{GQWR} Any pragmas which modify the lexical structure will be documented here.

R_{CSQC} There are no defined pragmas that change the lexical structure.

2.4 Annotations

R_{BGGC} Annotations begin with a `@` (at symbol) character.

Chapter 3
Builtin Types

Listing 3.1: Definition of ty declaration

```
ty ::= identifier
     | "boolean"
     | "integer" constraint_opt
     | "real"
     | "string"
     | "bit"
     | "bits" "(" bitwidth ")" bitfields_opt
     | "enumeration" "{" identifier_trailing_list "}"
     | "(" ty_list ")"
     | "array" "[" expr "]" "of" ty
     | "record" fields_opt
     | "exception" fields_opt

ty_list ::= ty "," ty_list
          |
          | ty

bitwidth ::= expr
           | "-" ":" ty
           | constraint

constraint ::= "{" constraint_range_list "}"

constraint_range ::= expr
                   | expr ".." expr

constraint_range_list ::= constraint_range "," constraint_range_list
                        | constraint_range

identifier_trailing_list ::= identifier_trailing "," identifier_trailing_list
                           | identifier_trailing
```

```
                              | identifier_trailing ","
```

I_{BYVL} Types describe the allowed values of variables, constants, function arguments, etc.

I_{JRDL} ASL allows the declaration of user named types. See 4.3 *Named type declarations*

I_{WYKZ} See also 7.1 *Type nomenclature* for definitions of some of the terminology used in this section.

3.1 Singular and aggregate types

D_{PQCK} Types are categorized as either *singular* or *aggregate*.

D_{NZWT} The builtin singular types are:

- `integer`
- `real`
- `string`
- `boolean`
- `bits`
- `bit`
- `enumeration`

D_{KNBD} The builtin aggregate types are:

- `tuple`
- `array`
- `record`
- `exception`

R_{GVZK} A named type is singular if it has the structure of a singular type, otherwise it is aggregate.

3.2 Restrictions on anonymous types

R$_{\texttt{GRVJ}}$ Anonymous enumeration, record and exception type declarations are not permitted except in named type declarations.

3.3 Integer type

The `integer` type represents mathematical integer values.

$\mathrm{I}_{\mathrm{HJBH}}$ There is no bound on the minimum and maximum integer value that can be represented.

$\mathrm{R}_{\mathrm{GWCP}}$ The syntax

```
integer <constraint>
```

denotes a *constrained integer* whose constraint is the set of values in the `constraint`.

For more details on constrained integers, see 7.4 *Constrained Integers*.

3.4 Real type

D_{CQXL}	The `real` type represents a rational number.
R_{XCJD}	There is no bound on the minimum and maximum absolute real value that can be represented.
I_{YFTF}	There is no mechanism in the language to generate an irrational value of `real` type.
I_{JQPK}	Conversions from `integer` to `real` are performed using the function `Real`
I_{WJCL}	Conversions from `real` to `integer` are performed using the function `RoundDown` or `RoundUp`.

3.5 String type

The `string` type represents strings of characters.

$\mathrm{I}_{\mathrm{DMNL}}$ Strings play relatively little role in specifications and the only operations on strings are equality and inequality tests.

3.6 Enumeration types

$\mathrm{D}_{\mathrm{YZBQ}}$ The `enumeration` type declaration defines a list of enumeration literals which act as global constants that can be compared for equality and inequality. The type's domain is the set of enumeration literals.

$\mathrm{R}_{\mathrm{DWSP}}$ No enumeration literal may explicitly appear in multiple `enumeration` type declarations.

$\mathrm{I}_{\mathrm{QMWT}}$ The use of 'explicitly' in rule DWSP clarifies that a named `enumeration` type may be used to specify the structure of another named `enumeration` type.

$\mathrm{R}_{\mathrm{HJYJ}}$ The type of an enumeration literal is the anonymous `enumeration` type which defined the literal.

$\mathrm{I}_{\mathrm{MZXL}}$ Note that enumeration literals exist in the same namespace as all other declared objects, including storage elements and subprograms, so no other declared object may have the same name in the same scope.

$\mathrm{I}_{\mathrm{PRPY}}$ Unlike many languages, there is no ordering defined for enumeration literals and therefore enumeration types do not support ordering comparisons such as `<=`.

Examples:

```
type SuperEnum of enumeration {LOW, HIGH};
// LOW and HIGH are of type enumeration {LOW, HIGH}

// legal
type SubEnum subtypes SuperEnum;

// legal
type OtherEnum of SuperEnum;

// illegal: no enumeration literal may appear in multiple enumeration type declarations.
type SubEnumIllegal1 of enumeration {LOW, HIGH} subtypes SuperEnum;

// illegal: enumeration {TOP, BOTTOM} does not subtype-satisfy SuperEnum of structure
    ↪enumeration {LOW, HIGH}
type SubEnumIllegal2 of enumeration {TOP, BOTTOM} subtypes SuperEnum;

// illegal: no enumeration literal may appear in multiple enumeration type declarations.
type ConflictingEnum of enumeration {LOW, HIGH};
```

3.7 Boolean type

The type `boolean` represents the algebraic boolean type. See also rule MXPS.

3.8 Bitvector type

D_{WXQV} The `bits(N)` type represents a bitvector of length `N`, where `N` may specify a fixed width or a constrained width.

U_{XKNL} There is no bound on the maximum bitvector length allowed, although an implementation may specify an upper limit. The minimum bound is zero. It is recognized that zero-length bitvectors might not be supported in systems to which ASL might be translated (such as SMT solvers), and an implementation might need to lower bitvector expressions to a form where they do not exist.

Example

```
func rotate{N}(src: bits(N), amount: integer) => bits(N)
begin
    let i = (amount MOD N) as integer {0..N-1};
    // upper may be a zero width bitvector
    let upper: bits(i)   = src[0+:i];
    let lower: bits(N-i) = src[i+:N-i];
    return [upper, lower];
end
```

3.8.1 Bitfields

Listing 3.2: Definition of bitfield declaration

```
bitfield_spec ::= ":" ty
                | bitfields_opt

bitfields_opt ::= "{" bitfield_list "}"
                |

bitfield_list ::= bitfield "," bitfield_list
                |
                | bitfield

bitfield ::= "[" slice_list "]" identifier bitfield_spec

slice ::= expr
        | expr ":" expr
        | expr "+:" expr
        | expr "*:" expr

slice_list ::= slice "," slice_list
             | slice
             | slice ","
```

I_{KGMC} The syntax for `bits` has an optional `bitfield_list` which allows bitslices of bitvectors to be treated as named fields which can be read or written.

R_{RMTQ} Each field of a bitvector type is defined using the bitfield syntax which specifies the name of the field and one or more valid bitslices of the bitvector which comprise that field.

R_{ZJSH} The type of a bitfield which does not have a type annotation is a bitvector type of the width of the bitfield.

R_{QCYM} The type of a bitfield with a type annotation must subtype-satisfy the bitvector type with size of the width of the bitfield.

I_{JDCC} Note that the wording "valid bitslice" means that the width and offsets of the slice must be correct under the rules for bitslices (5.9.1 *Bitslices*).

R_{MPMG} Bitfield names must be unique with respect to other fields in the same type.

I_{KPBX} Bitfield names are not part of the global or local scope, nor do they clash with other fields in bitvectors, records or exceptions.

R_{CGDG} The bitvector type may have bits in its bitvector representation which do not correspond to any bitfield.

$\mathrm{R}_{\texttt{CHBW}}$ The bit-slices of different bitfields may overlap.

$\mathrm{R}_{\texttt{CNHB}}$ The bit-slices of a single bitfield must not overlap.

$\mathrm{R}_{\texttt{GQVZ}}$ Fields of a bitvector type variable `myBits` can be read or written independently using the syntax `myBits.f` to refer to field `f` of `myBits`.

$\mathrm{R}_{\texttt{MXYQ}}$ Reads and writes of a bitvector type variable's field are treated as though they were of the field's type The bits of the field are mapped to the bits of the bitvector as though the slices comprising the field were concatenated in the order declared in the bitfield.

$\mathrm{R}_{\texttt{BDJK}}$ The width of each slice in a bitfield must be a non-negative, statically evaluable integer expression (including zero).

$\mathrm{R}_{\texttt{LGHS}}$ The offset of each slice in a bitfield must be a non-negative, statically evaluable integer expression (including zero).

$\mathrm{I}_{\texttt{XPDT}}$ Note that rule LGHS applies to bitfields, not bit slice expressions.

$\mathrm{R}_{\texttt{YYPN}}$ Two anonymous bitvector types are identical if they have the same width and they have bitfields with the same names and constituent bits, irrespective of the expressions used in their definition.

$\mathrm{I}_{\texttt{BGHB}}$ When statically determining whether two bitvector types have the same width, only statically evaluable expressions need be considered. See also 7.14 *Bitvector width comparison*.

$\mathrm{I}_{\texttt{FZMS}}$ A corollary of rule YYPN is that a bitvector declared with an empty `bitfield_list` is identical to a bitvector declared with no bitfields. E.g. `bits(8)` and `bits(8){}` are the same type.

$\mathrm{I}_{\texttt{CVXB}}$ Note that a named type whose *structure* is a bitvector type is not itself a bitvector type and is therefore not identical to any other bitvector type. See also 7.1.2 *Structure of a type*.

$\mathrm{I}_{\texttt{QDHP}}$ Note that whether or not a bit corresponds to a bitfield, it can be referenced using a bitslice expression.

Example of a bitvector type with bitfields

The following code declares a global variable whose type is a bitvector with bitfields.

```
var myData: bits(16) {[4]        flag,
                      [3:0, 8:5] data,
                      [9:0]      value};
```

- The expression `myData.flag` evaluates to the value `myData[4]` of type `bits(1)` (rules GQVZ and MXYQ)
- The expression `myData.data` evaluates to the value `[myData[3:0], myData[8:5]]` of type `bits(8)` (rules GQVZ and MXYQ)
- There is no bitfield which accesses `myData[15:10]` (rule CGDG)
- The `value` field overlaps with the other fields (rule CHBW)
- The slices `3:0` and `8:5` which define `data` do not overlap (rule CNHB)

Note that in the `data` bitfield, bits `3:0` come before bits `8:5` which is a different order from their occurrence in `myData`.

3.8.1.1 Nested bitfields

Bitfields may have nested bitfields. This can have several uses, one of which being being able to define two different views of a register.

Example

```
type CPTR_EL2_Type of bits(64) {
// common across formats
[31] TCPAC,
[30] TAM,

    // View when E2H register has value '0'
    [29:0] E2H0 {
        [20] TTA,
        [10] TFP,
        [8]  TZ
    },

    // View when E2H register has value '1'
    [29:0] E2H1 {
        [28]    TTA,
        [20+:2] FPEN,
        [16+:2] ZEN
    }

};

var E2H: bit;
var CPTR_EL2: CPTR_EL2_Type;

// Select TTA depending on the value of E2H
let TTA: bit = if E2H == '0' then CPTR_EL2.E2H0.TTA else CPTR_EL2.E2H1.TTA;
```

3.9 Array types

D

D_{DPXJ} The syntax `array [ expr ] of ty` declares a single dimensional array of `ty` with an index type derived from the expression `expr`.

R_{YHNV} This expression must be either:

- a non-negative, statically evaluable constrained integer expression `expr`, in which case the array has that many indices starting from 0
- the name of a type which has the structure of an enumeration type, in which case the array's indices are the enumeration literals of that type

R_{PXRR} If `expr` is of type `ty` which has the structure of a well-constrained integer whose domain contains only one value then the array is a *fixed size array* with number of indices equal to the value in the domain of `ty`.

R_{DGJT} If `expr` is of type `ty` which has the structure of a well-constrained integer whose domain contains more than one value then the array is a *constrained size array* with number of indices equal to `expr` and constrained as per the constraint on `expr`.

R_{HHCD} If `expr` is of type `ty` which has the structure of the under-constrained integer then the array is an *under-constrained array* with number of indices equal to `expr`.

Example:

```
// Declare an array of reals from arr1[0] to arr1[3]
type arr1 of array [4] of real;
// Declare an array with two entries arr2[big] and arr2[little]
type labels of enumeration {big, little};
type arr2 of array [labels] of bits(4);
```

R_{JJCJ} Array elements can be modified.

3.10 Tuple types

$\mathrm{R}_{\mathrm{TVPR}}$ Types can be combined into tuple types whose values consist of tuples of values of those types.

$\mathrm{I}_{\mathrm{MQWB}}$ For example, the expression `(TRUE, Zeros(32))` has type `(boolean, bits(32))`.

$\mathrm{R}_{\mathrm{CGWR}}$ A tuple type must contain at least 2 elements.

$\mathrm{R}_{\mathrm{JHKL}}$ The value and type of tuple elements cannot be modified.

3.11 Record types

Listing 3.3: Definition of field declaration

```
field ::= identifier ":" ty

field_list ::= field "," field_list
             |
             | field
```

D_{WGQS} A record is a structured type consisting of a list of field identifiers which denote individual storage elements.

R_{DXWN} A record type is described using the syntax `record {field_list}` where each element of the `field_list` specifies the name and type of the record's field identifiers.

R_{WFMF} The syntax `record` (with no `field_list`) is syntactic sugar for `record {}`.

R_{DLXV} Fields of a record can be read or written independently using the syntax `r.f` to refer to field `f` of a record `r`.

R_{MDZD} Record field names must be unique with respect to other fields in the same type.

I_{TFPS} Record field names are not part of the global or local scope, nor do they clash with other fields in bitvectors, records or exceptions.

Record values may be constructed as described in 5.5 *Record Expressions*.

3.12 Exception types

D$_{\text{QXYC}}$ An exception is a structured type consisting of a list of field identifiers which denote individual storage elements.

R$_{\text{MGHV}}$ An exception type is described using the syntax `exception {field_list}` where each element of the `field_list` specified the name and type of the exception's field identifiers.

R$_{\text{KGXL}}$ The syntax `exception` (with no `field_list`) is syntactic sugar for `exception {}`.

R$_{\text{BWDX}}$ Fields of an exception can be read or written independently using the syntax `r.f` to refer to field `f` of an exception `r`.

Exception values may be constructed as described in 5.6 *Exception Expressions*.

Example

```
type BAD_OPCODE of exception;

type UNDEFINED_OPCODE of exception {reason: string, opcode: bits(16)};

func test()
begin
  throw UNDEFINED_OPCODE{reason="Undefined", opcode='0111011101110111'};
end
```

R$_{\text{CHKR}}$ Exception field names must be unique with respect to other fields in the same type.

I$_{\text{HLBL}}$ Exception field names are not part of the global or local scope, nor do they clash with other fields in bitvectors, records or exceptions.

3.13 Named Types

R_{MHWM} A named type declaration declares an identifier, associated with a new type, with the structure and domain of values of a given base type. See 4.3 *Named type declarations*.

Named types allow the use of strong typing since a named type is not considered to be the same as any other type, including the type it derives its structure and domain from. Unlike some other languages, ASL does not provide a way to create an alias for an existing type. See also Chapter 7 *Type inference and type-checking*.

Chapter 4
Declaration syntax

D_{NMFP} A *declaration* introduces a new global object into the specification:

- named types,
- variables, let identifiers, constants and configs,
- functions, procedures, getters and setters.

Listing 4.1: Definition of declaration

```
decl ::= annotation decl
       | type_decl
       | storage_decl
       | function_decl
       | getter_decl
       | setter_decl
       | "pragma" identifier null_or_expr_list ";"
```

4.1 Scope of global declarations

R_{DHRC} Global identifiers are in scope everywhere.

I_{LWQQ} Since global identifiers are in scope everywhere, the order of declaration of global identifiers is not considered when determining their scope. Since declarations are also not allowed to be cyclical a valid sequence for declaring and initializing the objects can be determined by the implementation.

R_{HYQK} There is a single global namespace for all globally declared identifiers apart from setters. This namespace associates each identifier with the kind of global object to which it may refer.

R_{SWVP} An identifier which may refer to getters may also refer to setters.

I_{MVNZ} The existence of a single namespace means that:

- all global declarations must use unique names.
- local identifiers, including parameters of subprograms, cannot use any identifier that is used for global names.

I_{HJRD} Since the name of a setter must exist in the global namespace as the name of a similar getter (see 4.5.3 *Getters and Setters*) it is not possible to declare anything (other than a getter or setter) with the same name as a setter.

4.2 Compile-time and execution-time

I_{HSWW} Declarations in ASL may be described as either

- *compile-time-constant* or *non-compile-time-constant*
- *execution-time* or *non-execution-time*

See the following sections for usage and definitions: 4.4 *Global storage elements*, 6.2 *Local storage elements*, 8.3.1 *Execution-time expressions*, 8.3.2 *Compile-time-constant expressions*, 8.2.1 *Execution-time subprograms* and 8.2.2 *Compile-time-constant subprograms*.

`config` storage elements are both non-compile-time-constant and non-execution-time.

$\mathrm{R}_{\mathrm{PDLM}}$ A tool may optionally allow the initializer expression of `config` storage elements to be overridden as if the original initializer expression in the specification was replaced with the new initializer expression.

$\mathrm{R}_{\mathrm{FQLB}}$ The initializer of `config` storage elements may only be overridden before any execution-time initializer expressions or subprograms are evaluated.

4.3 Named type declarations

Listing 4.2: Definition of named type declaration

```
type_decl ::= "type" identifier "of" ty "subtypes" ty ";"
            | "type" identifier "of" ty ";"
            | "type" identifier "subtypes" ty with_opt ";"

with_opt ::= "with" fields_opt
           |
```

R_{KDKS} The type declaration `type id of ty ;` declares a new type `id` which has the type of `ty`

R_{YBWY} Named type declarations must not be recursive.

Example

```
type T1 of integer;         // the named type "T1" whose structure is integer
type T2 of (integer, T1);   // the named type "T2" whose structure is (integer, integer)
```

The following code declares two unique types with the same structure. Note that the two types are not related in any way and are not interchangeable. See also 7.6.2 *Type-Satisfaction*.

```
type qualifiedData of bits(16) { [4]        flag,
                                 [3:0, 8:5] data,
                                 [9:0]      value };
type DatawithFlag of qualifiedData;
```

In the following code, `TypeC.f` and `TypeD.f` have the same type: `TypeB` (and not `integer`).

```
type TypeA of integer;
type TypeB of TypeA;
type TypeC of record { f: TypeB };
type TypeD of TypeC;

func foo(x: TypeA)
begin
    pass;
end

func bar(x: TypeB)
begin
    pass;
end

func baz(x: integer)
begin
    pass;
end

func main() => integer
begin
    var x: TypeC;
    foo(x.f); // illegal -- x.f is of type TypeB which does not type-satisfy TypeA.
    bar(x.f); // legal
    baz(x.f); // legal -- TypeB type-satisfies integer
    var y: TypeD;
    foo(y.f); // illegal -- y.f is of type TypeB which does not type-satisfy TypeA.
    bar(y.f); // legal
    baz(y.f); // legal -- TypeB type-satisfies integer
    return 0;
end
```

The following type declarations are recursive and therefore illegal.

```
type base of record {one: other};

type other of base;
```

4.3.1 Subtypes

A named type may be a *subtype* of other types which allows it to be used in certain contexts where its supertype is expected in accordance with the rules of Chapter 7 *Type inference and type-checking*.

R_{ZRKM} The type declaration `type id1 of ty subtypes id2;` declares a new named type `id1` which has the same structure as `ty` and is also a subtype of the named type `id2`.

R_{ZWHP} The syntax `type id1 subtypes id2 ;` is syntactic sugar for `type id1 of id2 subtypes id2 ;`.

R_{LPDL} If `type id1 of ty subtypes id2;` is declared, it is an error if the named type `id2` is not declared.

R_{HQZY} The syntax `type id1 subtypes id2 with {new-fields...}` declares a new named type `id1` which has the structure of `id2`, with additional fields. `id1` is also a subtype of `id2`. `id2` must be a record or exception type. The declaration is equivalent to `type id1 of record {existing-id2-fields, new-fields...} subtypes id2`.

R_{SRHN} A named type `id1` may only be a subtype of a named type `id2` if `id1` subtype-satisfies `id2`.

R_{NXRX} The subtype relation is a partial order.

I_{KGKS} Since the subtype relation is reflexive, every type is also a subtype and supertype of itself.

I_{MTML} Since the subtype relation is transitive, if A is a subtype of B and B is a subtype of C then A is a subtype of C.

I_{JVRM} Since the subtype relation is antisymmetric it is an error if both `id1` is a subtype of `id2`, and `id2` is a subtype of `id1`.

I_{CHMP} All subtype relations (other than the transitive relationship) must be explicitly declared.

I_{LRVN} See 7.6.1 *Subtype-satisfaction* for the definition of subtype-satisfaction.

Examples

```
// Declare some named types
type superInt of integer;
type subInt of integer subtypes superInt ;
type uniqueInt of superInt;

func assign()
begin
    // Integer is subtype-satisfied by all the named types,
    // so it can be assigned to them by the assignment and
    // initialization type checking rules
    var myInt: integer;
    var mySuperInt : superInt  = myInt;
    var mySubInt   : subInt    = myInt;
    var myUniqueInt: uniqueInt = myInt;

    // Integer is subtype-satisfied by all the named types,
    // so it can be assigned from them by the assignment and
    // initialization type checking rules
    myInt = mySuperInt;
    myInt = mySubInt;
    myInt = myUniqueInt;

    // superInt is not a subtype of anything (apart from itself)
    // so it cannot be assigned to any other named type
    // Illegal: mySubInt    = mySuperInt;
    // Illegal: myUniqueInt = mySuperInt;

    // subInt is a subtype of superInt, so the assignment and
    // initialization type checking rules permit the following:
    mySuperInt = mySubInt;
    // But subInt and uniqueInt are not subtype related
    // so do not type-satisfy each other.
    // Illegal: myUniqueInt = mySubInt;

    // uniqueInt has no related subtype or supertype
    // so it cannot be assigned to any named type
    // Illegal: mySuperInt = myUniqueInt;
    // Illegal: mySubInt = myUniqueInt;
end
```

See 7.7.4 *Assignment and initialization type checking* for the assignment and initialization type checking rules.

```
type aNumberOfThings of integer;
type ShapeSides      of aNumberOfThings;
type AnimalLegs      of aNumberOfThings;
type InsectLegs      of integer subtypes AnimalLegs ;

func subtyping()
begin
    var  myCircleSides: ShapeSides = 1;              // legal
    var  myInt        : integer    = myCircleSides; // legal
    var  dogLegs      : AnimalLegs = myCircleSides; // illegal: unrelated types
    var  centipedeLegs: InsectLegs = 100;           // legal
    var  animalLegs   : AnimalLegs = centipedeLegs; // legal
    var  insectLegs   : InsectLegs = animalLegs;    // illegal: subtype is wrong way
end
```

```
type Coord2 of record { x: integer, y: integer };

// Declare a subtype, extended with a new field:
type Coord3 subtypes Coord2 with { z: integer };

// The above declaration is equivalent to:
type Coord3 of record { x: integer, y: integer, z: integer } subtypes Coord2;

// Invalid: field name 'x' is repeated:
type CoordX subtypes Coord2 with { x: real };
```

4.4 Global storage elements

Listing 4.3: Global variable declaration

```
storage_decl ::= "var" identifier ":" ty ";"
               | "var" identifier ty_opt "=" expr ";"
               | "let" identifier ty_opt "=" expr ";"
               | "constant" identifier ty_opt "=" expr ";"
               | "config" identifier ty_opt "=" expr ";"
```

This provides a way to define global *variable*, *let*, *constant* and *config* values.

R_{TRDJ} The `var` keyword is used to declare *global variable identifiers* denoting global storage elements which are all of the following: non-compile-time-constant, execution-time, mutable.

R_{DBZZ} The `let` keyword is used to declare *global let identifiers* denoting global storage elements which are all of the following: non-compile-time-constant, execution-time, immutable.

R_{BHMY} The `config` keyword is used to declare *global config identifiers* denoting global storage elements which are all of the following: non-compile-time-constant, non-execution-time, immutable.

I_{RBZW} Although `config` declares immutable identifiers, tooling may allow their value to be overridden before any execution-time initializer expressions or subprograms are evaluated, hence `config` declares elements which are both non-compile-time-constant and also non-execution-time.

R_{SBLX} The `constant` keyword is used to declare *global constant identifiers* denoting global storage elements which are all of the following: compile-time-constant, non-execution-time, immutable.

R_{ZNTH} Global bitvector storage elements shall have a determined width which is a non-execution-time expression.

R_{KSQP} A global let, constant or config storage element must be initialized with an initializer expression.

R_{WZJQ} If a global storage element is initialized with an initializer expression then the type of the global storage element may be omitted, in which case the type of the global storage element is the type of the initializer expression.

R_{TTMQ} The declarations, and any associated initialization expressions, for global storage elements must not include cycles.

Examples:

```
// example 1, invalid declaration of two variables with a cycle via the initialization
    ↪expressions
var a = b;
var b = a;


// example 2, and invalid declaration of 'var1' due to a cycle in its type specification
let size1 = Len(var1);    // function Len() returns the width of var1
var var1 : bits(size1);   // cycle -- the type of var1 depends on size1 which depends on var1
```

I_{VQHQ} See also 6.2 *Local storage elements*, 4.4.1 *Initialization of globals* and 6.2.2 *Initialization of locals*.

4.4.1 Initialization of globals

R_{GBNC} A global variable identifier is initialized with an arbitrary value of the variable's type if no initialization expression is given.

I_{GLHK} When initializing global variables without initializers, different values may be used for different variables, different values may be used for the same variable in different implementations, and different values may be used for the same variable in the same implementation in different executions. An implementation may, but is not required to, generate initial values using a pseudorandom generator whose seed is provided as an input parameter.

R_{FPMT} A global variable identifier is initialized with the value of the initialization expression if an initialization expression is given. The initialization expression is evaluated when the global variable identifier is initialized.

R_{FKGP} The initialization expression in a global `constant` declaration must be a compile-time-constant expression.

R$_{\texttt{DLXT}}$ A global `constant` identifier is initialized before any non-compile-time-constant initialization expressions or subprograms are evaluated.

R$_{\texttt{PRZN}}$ The initialization expression in a global `config` declaration must be a non-execution-time expression.

R$_{\texttt{RZLL}}$ A global `config` identifier is initialized after any compile-time-constant initializer expressions are evaluated and before any execution-time initializer expressions or subprograms are evaluated.

R$_{\texttt{CKGP}}$ A global `let` or `var` identifier is initialized after any non-execution-time initializer expressions are evaluated and before any execution-time subprograms are evaluated.

I$_{\texttt{QMQX}}$ Tools may provide a mechanism to override the initialization expression in a global `config` declaration.

R$_{\texttt{DJMC}}$ Initialization expressions in global declarations must not be side-effecting.

R$_{\texttt{PLYX}}$ Where the initialization expression in a `variable_declaration` is a bitvector of determined width, if the initialization expression type satisfies the declared type, then the declaration creates a storage element whose determined width is the determined width of the initialization expression.

I$_{\texttt{VYLK}}$ In the following contrived example, the declared type of `only` is a constrained width bitvector of undetermined width. However, its initialization expression is a constrained width bitvector of determined width `bits(1)`, so the resulting storage element is the constrained width bitvector of determined width `bits(1)` by rule PLYX.

```
var only: bits(-: integer{1,2}) = Zeros(1);
// Type-checker knows: width of only == 1
// only is of "determined width 1"
```

Examples:

```
var PC: bits(32);
constant MaxIrq: integer = 480;
var _R: array [16] of bits(32);
```

4.5 Subprogram declarations

D_{JWKG} Functions, procedures, getters and setters are collectively referred to as *subprograms*.

I_{DVSM} In this section we provide rules for all four kinds of subprogram, with more specific rules in subsequent sections.

I_{GKLW} Multiple subprograms may be defined with the same name provided that each definition has a different number or type of arguments to allow disambiguation. See 7.6.4 *Subprogram clashing*.

R_{KFGJ} The optional `parameter_list` declares the parameters of the subprogram. ASL permits the declaration of subprograms with formals which are parameterized such that where a formal or return type is a bitvector, its width may depend on the value of the subprogram parameters. For details see 7.8.1 *Dependently typed bit vector formals*.

R_{PTDD} The `formal_list` declares the formal arguments of the subprogram. Each element of the `formal_list` of a subprogram declaration declares a *let identifier* of the given name and type in the scope of the subprogram body, denoting a local storage element which is all of the following: non-compile-time-constant, execution-time, immutable.

R_{JBXS} If part of the type of any formal argument is a bitvector whose width is not a compile-time-constant, then any identifiers used in the bitvector's width expression must be included in exactly one of the `formal_list` or the `parameter_list` and are declared as *let identifiers* in the scope of the subprogram body, denoting a local storage element which is all of the following: non-compile-time-constant, execution-time, immutable.

D_{QMYP} The type of a subprogram is its *signature* and consists of its return type (if any) and the types of its formal arguments.

D_{JWXX} A *control flow path* is a path through the control flow graph derived from the syntactic structure of a subprogram. Where a conditional branch occurs (e.g. in an `if` or `case` statement) the condition is ignored; all branches are added to the control flow graph.

D_{VFTV} An *always-throw procedure* is one in which all control flow paths terminate with a throw statement or a call to an always-throw procedure. Members of a set of mutually recursive procedures in which all control flow paths terminate with a throw statement, a call to a member of the set, or a call to an always-throw procedure outside the set are always-throw.

R_{WKHC} All control flow paths in a function or getter must terminate with a return statement, a throw statement, a call to `Unreachable()` or a call to an always-throw procedure.

R_{DFWZ} It is not an error for execution of a procedure or setter to end without a return statement.

R_{QCVM} When a subprogram invocation is executed, the actual arguments of the invocation are evaluated, and the resulting *actual values* are used to initialize the corresponding identifiers declared in the formal argument list. Any bitvector width parameters which are not also formal arguments take their value from the width of the related formals. The order of evaluation of arguments is unspecified (see 8.5.1 *Evaluation order*).

I_{TWJF} Note that all subprogram arguments are passed by value (not by reference) and that bitvector formal arguments may be dependently typed.

I_{PFNG} It is recommended that definitions sharing the same name are kept contiguous in specifications to further help humans to disambiguate.

R_{HDGV} It is an error to declare a subprogram formal argument or parameter with the same name as a global variable (see also 4.1 *Scope of global declarations*).

R_{KCMK} It is an error to provide multiple definitions with the same argument types for the same function or procedure, even if those definitions are identical.

4.5.1 Side-effect-free subprograms

D_{GWWP} A *side-effect-free subprogram* is one that does not mutate global storage elements.

R

R_{SFPM} A side-effect-free subprogram must not contain an assignment statement whose left hand side refers to a mutable global storage element, or a call to a subprogram that is not side-effect-free, or a throw statement.

I_{BCWW} A side-effect-free subprogram may read mutable global storage elements, and may call other side-effect-free subprograms. It may declare local variables and assign to them.

I_{FYFN} A side-effect-free subprogram may contain `assert` statements, calls to the `Unreachable()` subprogram and calls to the `print` subprogram.

4.5.2 Functions and procedures

```
function_decl ::= "func" identifier parameters_opt "(" formal_list ")" return_ty_opt
    ↪subprogram_body

formal ::= identifier ":" ty

formal_list ::= formal "," formal_list
              |
              | formal

parameters_opt ::= "{" parameter_list "}"
                 |

parameter ::= identifier ty_opt

parameter_list ::= parameter "," parameter_list
                 |
                 | parameter

return_ty_opt ::= "=>" ty
                |

subprogram_body ::= "begin" stmt_list "end"
```

R_{RXHX} The `func` keyword declares a subprogram. If a return type is included it is a *function declaration*, otherwise it is a *procedure declaration*.

R_{JGVX} Functions must only be used in function invocation *expressions*. See also 7.8.2 *Subprogram invocations*.

R_{FRDX} Procedures must only be used in procedure call statements.

Example:

```
// Function 'Add'
func Add{N}(x: bits(N), y: bits(N)) => bits(N)
begin
    return x + y;
end

var Counter: integer = 0;

// Procedure 'IncrementCounter'
func IncrementCounter(inc: integer)
begin
    Counter = Counter + inc;
    return;
end
```

4.5.3 Getters and Setters

```
getter_decl ::= "getter" identifier parameters_opt args_opt "=>" ty subprogram_body

setter_decl ::= "setter" identifier parameters_opt args_opt "=" identifier ":" ty
    ↪subprogram_body

args_opt ::= "[" formal_list "]"
           |
```

R$_{MZJJ}$ The `getter` keyword introduces a *getter declaration*.

R$_{MBZP}$ The `setter` keyword introduces a *setter declaration*.

I$_{QJNF}$ Getters are function-like subprograms whose invocation syntactically looks like a variable read or array read. See also rule ZFFV in section 5.9 *Arrays, bitslices and invoking getter functions*.

I$_{FBVH}$ Setters are procedure-like subprograms whose invocation syntactically looks like a variable write or array write. See also rule YDFQ in section 6.5.3 *Setter invocation*.

I$_{TVDT}$ Getters and setters are frequently associated with storage where the storage accessed depends on the current privilege/security mode of the processor.

R$_{QKXV}$ The optional `parameter_list` declares the parameters of the getter or setter. ASL permits the declaration of subprograms with formals which are parameterized such that where a formal or return type is a bitvector, its width may depend on the value of the subprogram parameters. For details see 7.8.1 *Dependently typed bit vector formals*.

R$_{NCTB}$ The optional `formal_list` declares the formal arguments of the getter or setter. Each element of the `formal_list` of a subprogram declaration declares a *let identifier* of the given name and type in the scope of the subprogram body, denoting a local storage element which is all of the following: non-compile-time-constant, execution-time, immutable.

D$_{MRYB}$ The *RHS argument* of a setter declaration is an argument whose name and type are given in a `setter_declaration` following the `=` symbol.

D$_{BHPJ}$ The formal arguments of a setter are any items in the `formal_list` of the setter's declaration followed by the RHS argument.

I$_{WHLV}$ The formal arguments of a getter or setter are denoted using square brackets `[` and `]` rather than parentheses `(` and `)`. The square brackets may be omitted if the getter or setter has no formal arguments (beyond the RHS argument of a setter).

R$_{LCSZ}$ If a setter function is defined, a similar getter function must be declared such that the getter may be invoked with the same actual arguments as declared in the setter's `formal_list`.

Examples:

```
getter SP[] => bits(64)
begin
    return _SP[CurrentEL()];
end

setter SP[] = value: bits(64)
begin
    _SP[CurrentEL()] = value;
    return;
end
```

R$_{TJRH}$ If a getter function is defined without an argument list, no other getter functions of the same name are allowed.

Example:

```
getter Foo => bits(8)
begin
    return _foo;
end

// Not allowed, getter of name Foo with no argument list is defined above
getter Foo[n: integer] => bits(8)
begin
    return _foo1[n];
end

getter Bar[] => bits(8)
begin
    return _bar;
```

```
end

// Allowed, getter of name Bar with argument list is defined
getter Bar[n: integer] => bits(8)
begin
    return _bar1[n];
end
```

4.6 Pragmas

I_ZGJQ The pragma statement provides information or commands to tools consuming the specification.

Listing 4.4: Definition of pragma

```
decl:pragma ::= "pragma" identifier null_or_expr_list ";"
```

D_QTJC All pragma identifiers starting with “asl_” are referred to as *ASL language pragmas*. Others are referred to as *tool specific pragmas*.

R_GBNH The effect and scope of a pragma is dependent on the pragma identifier.

R_PMHS Pragma identifiers are independent of other identifiers.

R_GFSD Pragmas can take zero or more parameters.

R_BWYF ASL language pragmas are reserved and may be specified in future ASL versions.

Example:

```
pragma asl_pragma1;
pragma asl_op 1, "start";
func my_function_with_pragma()
begin
    pass;
end
```

R_FPPF If the effect of a tool specific pragma is required for the correct operation of the specification then the specification is not compliant to this standard.

R_VXCS For compliant specifications any tool specific pragmas not recognised by a tool which processes ASL must not stop the operation of the tool.

I_SLNQ It is recommended that tools which process ASL should warn users of unrecognised tool specific pragmas.

Example:

```
pragma my_tool_pragma1;
pragma other_tool_op '0010', 123 ;
func my_function_with_tool_pragmas()
begin
    pass;
end
```

4.7 Annotations

```
annotation ::= "@" identifier "(" null_or_expr_list ")"
```

Subprograms and statements can be annotated with code annotations. These annotations provide further metadata about how a declaration or statement should be interpreted or handled by an implementation.

4.7.1 Recursion limits

```
@recurselimit(<LIMIT>)
```

The `recurselimit` annotation annotates that a subprogram can recurse a maximum of `<LIMIT>` times, directly or indirectly.

A recursive path is a function call path that repeats.

R_{BYRT} At least one function in a recursive path must a `@recurselimit` annotation.

R_{VQRH} The recursion limit must be a non-execution-time integer expression.

R_{TWDQ} When the recursion limit is exceeded, an error should occur.

I_{FXJV} These annotations may be used by an implementation to unroll recursive code.

```
@recurselimit(4)
func foo(a: integer) => integer
begin
    return bar(b);
end

@recurselimit(3)
func bar(b: integer) => integer
begin
    return foo(b);
end
```

In the above example and longest possible call path is `foo->bar->foo->bar->foo->bar-foo->bar`. The final call to `bar` results in an error.

4.7.2 Loop limits

See 6.9.1 *Loop limits*

Chapter 5
Expression syntax

When reading this section, the definitions of 8.1.1 *Statically evaluable expressions* and 8.3 *Evaluation of expressions* are useful.

$\mathrm{D}_{\mathrm{KXWT}}$ The term *Primitive Operations* denotes the set of operations available in the expression syntax.

$\mathrm{I}_{\mathrm{HSQL}}$ This includes `binop`, `unop` and `if..then..else` expressions.

Expressions calculate values. All expressions have a unique type. The type of an expression can be a tuple. Expressions can have side effects and can raise exceptions and, therefore, there are constraints on the evaluation order and on the side-effects/exceptions to avoid surprising or unpredictable behavior (see 8.5.1 *Evaluation order*).

Expressions contain several levels of precedence. This is expressed in the grammar in the usual way by a set of mutually recursive definitions.

```
expr ::= "if" cexpr "then" expr elsif_expr_list "else" expr
       | cexpr
```

5.1 Conditional expressions

```
expr ::= "if" cexpr "then" expr elsif_expr_list "else" expr
       | cexpr

elsif_expr ::= "elsif" expr "then" expr
```

R$_{\text{YCDB}}$ A conditional expression evaluates to its *then* expression if the condition expression evaluates to `TRUE`. If the condition expression evaluates to `FALSE` each *elsif* condition expression is evaluated sequentially until an elsif condition expression evaluates to `TRUE`; the conditional expression evaluates to the corresponding elsif expression. If no elsif expression evaluates to `TRUE` the conditional expression evaluates to the *else* expression.

5.2 Binary and unary operators

```
cexpr ::= cexpr binop_boolean cexpr_cmp
        | cexpr checked_type_constraint
        | cexpr_cmp

cexpr_cmp ::= cexpr_cmp binop_comparison cexpr_add_sub
            | cexpr_add_sub

cexpr_add_sub ::= cexpr_add_sub binop_add_sub_logic cexpr_mul_div
                | cexpr_mul_div

cexpr_mul_div ::= cexpr_mul_div binop_mul_div_shift cexpr_pow
                | cexpr_pow

cexpr_pow ::= cexpr_pow binop_pow bexpr
            | bexpr

binop_boolean ::= "&&"
                | "||"
                | "-->"
                | "<->"

binop_comparison ::= "=="
                   | "!="
                   | ">"
                   | ">="
                   | "<"
                   | "<="

binop_add_sub_logic ::= "+"
                      | "-"
                      | "OR"
                      | "XOR"
                      | "AND"

binop_mul_div_shift ::= "*"
                      | "/"
                      | "DIV"
                      | "DIVRM"
                      | "MOD"
                      | "<<"
                      | ">>"

binop_pow ::= "^"

bexpr ::= unop bexpr
        | expr_term

unop ::= "-"
       | "!"
       | "NOT"

binop_in ::= "IN"

expr_term ::= expr_atom binop_in pattern_set
            | "UNKNOWN" ":" ty
            | expr_atom
```

ASL provides the usual arithmetic and logical operators as well as tests for set membership and for comparing values with `bitmask_lit`s. Note that, unlike C, the operator ^ represents exponentiation, not exclusive-or.

The primitive operators are defined in tables Table 7.1, Table 7.2, Table 7.3, Table 7.4, Table 7.5 and Table 7.6 and apply only to the types shown there.

5.2.1 Operator precedence

Operator precedence is used to disambiguate binary expressions. The operator classes and precedence order is defined by Table 5.1.

Given two binary operators op_1 and op_2, an expression of the form $x\ op_1\ y\ op_2\ z$, is interpreted as $(x\ op_1\ y)\ op_2\ z$ if op_1 has higher precedence than op_2 or as $x\ op_1\ (y\ op_2\ z)$ if op_1 has lower precedence than op_2. If op_1 is associative and $op_1 = op_2$ then the expression may be interpreted as either $(x\ op_1\ y)\ op_2\ z$ or as $x\ op_1\ (y\ op_2\ z)$ since there is no difference. Otherwise, an operator precedence error is reported.

The following operators are associative: `+ * && || AND OR XOR`

Table 5.1: Precedence classes

Precedence	Class	Operators
1 (Highest)	**Membership**	`IN`
2	**Unary**	`- ! NOT`
3	**Power**	`^`
4	**Mul-Div-Shift**	`* / DIV DIVRM MOD << >>`
5	**Add-Sub-Logic**	`+ - AND OR XOR`
6	**Comparison**	`== != > >= < <=`
7 (Lowest)	**Boolean**	`&& || --> <->`

Examples:

```
func operator_precedence(
    a: integer,
    b: integer,
    c: integer,
    d: bits(8),
    e: bits(8),
    f: bits(8),
    g: boolean)
begin

    let p_m_s = a * b - c;
    // '*' has higher precedence than '-' so interpreted as:
    let p_m_s_I = (a * b) - c;
    assert(p_m_s == p_m_s_I);

    let p_s_m = a - b * c;
    // '*' has higher precedence than '-' so interpreted as:
    let p_s_m_I = a - (b * c);
    assert(p_s_m == p_s_m_I);

    // let p_a_s = a + b - c;
    // '+' has equal precedence to '-' so causes a compile-time error.
    // Must be written as either:
    let p_a_s_A1 = (a + b) - c;
    let p_a_s_A2 = a + (b - c);

    // let p_s_a = a - b + c;
    // '-' has equal precedence to '+' so causes a compile-time error.
    // Must be written as either:
    let p_s_a_A1 = (a - b) + c;
    let p_s_a_A2 = a - (b + c);

    let p_a_e = a + b ^ c;
    // '^' has higher precedence than '+' so interpreted as:
    let p_a_e_I = a + (b ^ c);
    assert(p_a_e == p_a_e_I);

    let p_and_and = d AND e AND f;
    // 'AND' is associative so can be interpreted as either:
    let p_and_and_i1 = (d AND e) AND f;
    let p_and_and_i2 = d AND (e AND f);
    assert(p_and_and == p_and_and_i1);
    assert(p_and_and == p_and_and_i2);

    // let p_and_or = d AND e OR f;
    // 'AND' and 'OR' have no defined precedence so causes a compile-time error.
    // Must be written as either:
    let p_and_or_A1 = (d AND e) OR f;
    let p_and_or_A2 = d AND (e OR f);

    let p_band_eq = g && a == b;
    // '&&' is of precedence class 'Boolean'.
    // '==' is of precedence class 'Comparison'.
    // 'Comparison' has higher precedence than 'Boolean' so interpreted as:
    let p_band_eq_I = g && (a == b);
    assert(p_band_eq == p_band_eq_I);

    // let p_eq_eq = a == b == g;
    // '==' is not associative so causes a compile-time error.
    // Must be written as:
    let p_eq_eq_A1 = (a == b) == g;
    // Note: 'a == (b == g)' is not valid as it does not type satisfy.
end
```

5.3 Pattern matching

The binary operator `IN` tests whether a value (referred to as the discriminant) matches any item from a `pattern_set`. Lists of patterns are also used in case statements (6.8 *Case Statements*).

```
pattern ::= "-"
          | expr
          | expr ".." expr
          | "<=" expr
          | ">=" expr
          | pattern_set

pattern_set ::= "{" pattern_list "}"
              | "!" "{" pattern_list "}"
              | bitmask_lit

pattern_list ::= pattern "," pattern_list
               | pattern
```

A `pattern_set` consists of one of:

- a literal bitmask of the same length as the discriminant.
- a list of one or more patterns. These match if any member of the set matches the discriminant.
- a negated list of one or more patterns. These match if no members of the set matches the discriminant.

A pattern consists of one of:

- The '`-`' pattern, which always matches.
- A compile-time-constant expression of a type that has the structure of any one of the following primitive types: `boolean`, `integer`, `real`, `enumeration` or `bits(N)`.
- A literal bitmask of the same length as the discriminant.
- A range of integers or reals, with lower and upper bounds as `expr .. expr`, or an upper bound as `<= expr` or a lower bound as `>= expr`. Ranges are closed, meaning that they include their endpoints.
- A `pattern_set`.

Pattern matching also supports the concurrent matching of multiple discriminants.

- Multiple discriminants can be matched at once by forming a tuple of discriminants and a tuple used in the `pattern_set`. Both tuples must have the same number of elements. A successful pattern match occurs when each discriminant term matches the respective term of the pattern tuple.

Any expression which can be used as an initializer for a global constant, can be used in a pattern. This includes calls to compile-time-constant functions.

It is not permitted to use a function parameter or `let` or `var` variable in a pattern.

Individual bits of a bitmask can be specified as `x` to always match that bit with the associated bit of the discriminant.

I_{WLNM} There is no type for bitmasks (`bitmask_lit`). Bitmasks cannot be assigned to variables or passed into or out of functions.

R_{ZNDL} The `IN` operator is equivalent to testing its first operand for equality against each value in the (possibly infinite) set denoted by the second operand, and taking the logical OR of the result. Values denoted by a set of patterns comprise the union of the set of values denoted by each pattern. Values denoted by a `bitmask_lit` comprise all bitvectors that could match the bit-mask. It is not an error if any or all of the values denoted by the first operand can be statically determined to never compare equal with the second operand.

Examples:

```
    let expr_A = '111' IN {'1xx'};                   //  TRUE
    let expr_Aa = '111' IN '1xx';                    //  TRUE
    let expr_B = '111' IN {'0xx'};                   //  FALSE
    let expr_C = 3 IN {2,3,4};                       //  TRUE
    let expr_D = 1 IN {2,3,4};                       //  FALSE
    let expr_E = 3 IN {1..10};                       //  TRUE
    let expr_F = 3 IN {<= 10};                       //  TRUE
    let expr_G = 3 IN !{1,2,4};                      //  TRUE
    let expr_H = (1,'10') IN {(1,'1x')};             //  TRUE
    let expr_I = (1,'10') IN {(1,'0x'), (2, '1x')};  //  FALSE (see note below)

    // The last two expressions are equivalent to:
    let expr_H = (1 IN {1}) && ('10' IN {'1x'});
    let expr_I = ((1 IN {1}) && ('10' IN {'0x'})) ||
                 ((1 IN {2}) && ('10' IN {'1x'}));
```

Note that `expr_H` and `expr_I` use a tuple to simultaneously pattern match two values, the integer `1` and the bitvector `'10'`. For `expr_I` although these values each match parts of the tuple patterns in the `pattern_set`, they do not simultaneously match all elements of either of the tuple patterns `(1,'0x')` or `(2, '1x')`.

5.4 Atomic expressions

```
expr_atom ::= identifier
            | identifier "(" null_or_expr_list ")"
            | identifier "{" field_assignment_list "}"
            | literal_expr
            | expr_atom "[" null_or_slice_list "]"
            | expr_atom "." identifier
            | expr_atom "." "[" identifier_list "]"
            | "(" pattern_list ")"
            | "[" expr_list "]"

field_assignment ::= identifier "=" expr

field_assignment_list ::= field_assignment "," field_assignment_list
                        |
                        | field_assignment
```

Atomic expressions consist of:

- Local and global storage element identifiers. The value of such an expression is the value stored in the storage element declared with the given identifier, and has the type of the identifier.
- *5.7 Function Invocations*.
- *5.5 Record Expressions*.
- *5.8 Literal constants*.
- An indexed read. See *5.9 Arrays, bitslices and invoking getter functions*.
- Field `f` of a record or bitvector `r` is extracted using `r.f`.
- Multiple fields f_1, .. f_n of a bitvector r or bitvector fields of a record are extracted and concatenated using $r.[f_1, .., f_n]$.

  R_{RCSD} It is an error if any field used is not of type bitvector.
- *5.10 Tuples*.
- A list of slices (i.e., `i +: w`, `j : i`, `i *: w` or `i`) can be used to mean the concatenation of a list of individual slices. For example, `b[3:0, 7:4]` denotes the 8-bit bitvector consisting of the bottom nibbles of `b` in reverse order.

Examples:

```
    expr_A = Replicate(x, 4);    // function invocation
    expr_B = (N - 1);            // parenthesised expression
    expr_C = (TRUE, Zeros(32));  // 2-tuple
```

```
type pair of record {x: integer, y: integer};

func field_assignments() => pair
begin
    let p: pair = pair {x = 1, y = 2};  // record creation
    return p;
end
```

5.5 Record Expressions

```
expr_atom:structured ::= identifier "{" field_assignment_list "}"
```

$\mathrm{R}_{\texttt{WBCQ}}$ The `identifier` in a *record expression* must be a named type with the structure of a record type, and whose fields have the values given in the `field_assignment_list`.

$\mathrm{R}_{\texttt{DYQZ}}$ A record expression shall assign every field of the record.

5.6 Exception Expressions

```
expr_atom:structured ::= identifier "{" field_assignment_list "}"
```

R_{ZWCH} The `identifier` in an *exception expression* must be a named type with the structure of an exception type, and whose fields have the values given in the `field_assignment_list`.

R_{KCDS} An exception expression shall assign every field of the exception.

5.7 Function Invocations

```
expr_atom:invoke ::= identifier "(" null_or_expr_list ")"
```

A function invocation expression calls a defined function. Several functions may be defined with the same name, and the types of the arguments are used at compile-time to resolve to a specific function.

$\mathrm{I}_{\mathrm{VGSP}}$ Overload resolution is based only on the name and argument types, and takes no account of how the result is used.

$\mathrm{I}_{\mathrm{QSLR}}$ ASL pre-defines some standard functions. For a list, see Chapter 9 *Standard library*.

For more detail on how function invocations are resolved, see 7.8.2 *Subprogram invocations*.

5.8 Literal constants

```
literal_expr ::= int_lit
               | hex_lit
               | real_lit
               | bitvector_lit
               | string_lit
               | boolean_lit
```

R_{TSKH} Literal constants consist of literals as defined in the above grammar rule (see (2.2.2 *Literals*)), and enumeration literals.

R_{ZDSJ} Integer literals (both decimal and hexadecimal) have constrained integer type. The type of an integer literal is the constrained integer type whose constraint holds only the value of the literal.

R_{LLJZ} Real literals have type `real`.

R_{QNQV} String literals have type `string`.

R_{SPPT} The type of a bit-vector literal is `bits(N)` where `N` is the number of ‘0’ and ‘1’ characters in the literal.

I_{WDMD} Any spaces included in the literal do not affect the length of the literal so, for example, `'11110000'` and `'1111 ↪0000'` are both bit-vectors of type `bits(8)`.

R_{YTKY} Boolean literals have type `boolean`. See rule MXPS.

Examples:

```
    let expr_H = 1;                 //   integer
    let expr_I = 0xffff_ffff;       //   integer
    let expr_J = 2.0;               //   real
    let expr_K = "Hello World";     //   string
    let expr_L = '';                //   bits(0)
    let expr_M = '1110 0000';       //   bits(8)
    let expr_N = TRUE;              //   boolean
    type ET of enumeration{X,Y};
    let expr_ET = X;                //   ET
```

5.9 Arrays, bitslices and invoking getter functions

```
expr_atom ::= identifier
            | identifier "(" null_or_expr_list ")"
            | identifier "{" field_assignment_list "}"
            | literal_expr
            | expr_atom "[" null_or_slice_list "]"
            | expr_atom "." identifier
            | expr_atom "." "[" identifier_list "]"
            | "(" pattern_list ")"
            | "[" expr_list "]"

slice ::= expr
        | expr ":" expr
        | expr "+:" expr
        | expr "*:" expr

slice_list ::= slice "," slice_list
             | slice
             | slice ","

null_or_slice ::= slice

null_or_slice_list ::= null_or_slice "," null_or_slice_list
                     |
                     | null_or_slice
```

The same syntax is used in ASL for array indexing, bitslicing and invoking getter functions.

This can be one of:

- The `i`th element of an array `a` is extracted using `a[i]`.

R_{ZTRR} It is an error to use an index that is out of range.

- A getter function F can be applied to a list of arguments a_1, .., a_n using $F[a_1$, .., $a_n]$.

R_{DVVQ} If a getter function does not define any arguments but contains an empty argument list in the declaration, the getter must be invoked with an empty set of square brackets.

R_{GXQH} If a getter function does not contain an argument list in the declaration, the getter must not be invoked with square brackets.

Example:

```
getter foo[] => integer;
getter bar => integer;

func baz()
begin
    var a = foo[];  // Must provide []
    var a = bar;    // Must not provide []
end
```

- A bitslice of length `w` consisting of bits `i` up to and including `i+w-1` of a bitvector or integer `b` is extracted using `b[i +: w]`. The notation `b[j:i]` is syntactic sugar for `b[i +: j-i+1]`. The notation `b[i]` is syntactic sugar for `b[i +: 1]`. The notation `b[i *: n]` is syntactic sugar for `b[i*n +: n]` For example, `b[16 +: 8]`, `b[23 : 16]` and `b[2 *: 8]` all denote the 8-bit bitvector obtained by extracting bits from `16` up to and including `23`.

  Bitslicing an integer returns a slice of the twos-complement representation of the integer.

  It is an error to use a negative slice width or a slice that starts with a negative index or that extends beyond the length of a bitvector.

  A slice may result in a zero-length bitvector. For example, `b[0:1]` results in a zero-length bitvector.

Distinguishing between array indexing, bitslicing and invoking getter functions is based on whether there is a global variable or parameterized getter function in scope and on whether the type of the variable is a bitvector or an array.

R$_{ZFFV}$ Where an `expr_atom` consists of an identifier which is declared as a getter, then the `expr_atom` is treated in the same way as a function invocation. In this case, if a sequence of `null_or_slice_lists` is present, each must consist of a single expression. The sequence of `null_or_slice_lists` shall be used as the actual expressions for the invocation of the getter.

Examples:

```
    let expr_O = Mem[addr, 4];          //  Call getter function Mem
    let expr_P = _R[i];                 //  Read element i of array _R
    let expr_Q = result[63];            //  Extract bit 63 of bitvector result
    let expr_R = result[31:0];          //  Extract bits 31 down to 0 of bitvector result
    let expr_S = EPSR[26:25, 15:10];    //  Extract two bitslices and concatenate them
    let expr_T = PSTATE.[N,Z,C,V];      //  Extract four bitvector fields from PSTATE and
        ↪concatenate them
```

5.9.1 Bitslices

I$_{TFSZ}$ Bit vectors support bitslice operations that extract a number of contiguous or non-contiguous bits from a bit vector. A bitslice operation may result in a zero-length bitslice.

R$_{NHGP}$ The expressions specifying a bitslice must be such that the width of the resulting bitvector has constrained type.

For example, in `b[j:i]`, both `j` and `i` must have constrained type.

R$_{WZCS}$ The width of a bitslice must be any non-negative, statically evaluable integer expression (including zero).

I$_{KLDY}$ Because of the use of statically evaluable expressions (see 8.1.1 *Statically evaluable expressions*) in bitslices, the width of the bit vector resulting from the bitslice can be used for checking type-satisfaction (See also 7.6.2 *Type-Satisfaction*.)

R$_{KTBG}$ It is an error if any bits selected by a bitslice are not in range for the expression being sliced. If the offset of a bitslice depends on a statically evaluable expression then this shall be checked at compile time. Otherwise a bounds check will occur at execution-time and an implementation defined exception shall be thrown if it fails.

I$_{JEJD}$ If the offset of a bitslice depends on an expression whose type is a constrained integer, then the compiler may elide the execution-time bounds check if all possible offsets would be legal at runtime.

R$_{SNQJ}$ An expression or subexpression which may result in a zero-length bitvector must not be side-effecting.

The above rule is intended to allow implementations to transform expressions to a form where zero-length bitvectors do not exist.

Example of bitslice

```
func R_sl()
begin
    let offset: integer = f();
    let k: integer {3, 7} = getWid();
    var src: bits(k);
    var dst: bits(k-1);

    dst = src[offset+k-2:offset]; // legal
    // but requires an execution-time check that
    // offset+k-2 < k
    // offset >= 0

    dst = src[offset+:k-1]; // legal
    // but requires an execution-time bounds check that
    // offset+k-1 <= k
    // offset >= 0

    dst = src[k:k-offset];
    // illegal (bitslice non-mutable width requirement)
    // since width is mutable `offset+1`

    let w = offset;
    dst =  src[w:1];
    // illegal (type-satisfaction requirement)
    // although width is a statically evaluable expression
    // since (w != width of dst)

    dst[0+:w] = src[0+:w];
    // legal
    // but requires an execution-time bounds check that:
    // max index of LHS bitslice <= max index of dst
    // and
    // max index of RHS bitslice <= max index of src

    var zw: integer{0,1,2};
    var zb = f()[0+:zw];
    // legal, as long as f() is side-effect-free

    let d = [f()[1+:zw], g()[1 +: 8]];
    // f() must not be side-effecting as it is used in a sub-expression
    // that may produce a zero-with bitvector.
    // g() may be side-effecting as the sub-expression it is in
    // produces a non-zero-width bitvector.
    // The RHS expression as a whole may be side-effecting.
end
```

5.10 Tuples

R_{JPVL}

Tuple expressions are supported to allow multiple assignment and functions that return multiple values.

Examples:

```
// declaration of a tuple type
type MyTupleT of (boolean, integer);

// a function that returns a tuple
func calcEnable() => MyTupleT
begin
  return (TRUE, data);  // returning a 2-tuple
end

func example1() => (integer, integer)
  begin
  // declaration let identifier (of type tuple), with initialization
  let default_range : (integer, integer) = (0, 31);

  // declaration of two var identifiers
  // The type information for a and b is taken from the initialization expression
  var (a,b) = default_range;

  // declare a local identifier using a tuple type
  var enable_value : MyTupleT;

  // assignment to a tuple
  enable_value = calcEnable();


  var hi, lo :integer;
  // multiple assignment to the list of elements `lo, hi` from
  // a conditional expression that has one tuple literal `(0,63)`
  // and a variable default_range, of type tuple.
  // Note that the expression on the left side of the equals sign is
  // not a tuple but instead a list of elements.
  (lo, hi) = if sf then (0, 63) else default_range;

  return (lo, enable_value.item1);
end
```

R$_{\text{KVNX}}$

Individual elements of a tuple may be accessed read-only. Writing values to an individual element of a tuple is not supported.

I$_{\text{TSXL}}$

An expression to access an individual element of a tuple is a tuple expression followed by a dot (`.`) and then a specific identifier. This identifier must be the combination of the exact string `item` followed by an integer. The integer must be the digit 0, or a positive number (with no leading zeros).

Example:

```
(1, 2).item0     // the tuple `(1,2)`, the string `item`, the integer 0
```

I$_{\text{XFPV}}$

Tuple element selection is position-based. The first element is selected with `.item0`, the second with `.item1`, the third with `item2` and so on. It is invalid to specify an element beyond the size of the tuple.

Examples:

```
a = (1, 2).item0;  // a = 1;
b = (1, 2).item1;  // b = 2;
// the above 2 statements are equivalent to the following multi-assign statement.
(a, b) = (1, 2);
```

5.11 Checked type conversions

R$_{PBLF}$ The syntax

```
cexpr as ty
```

is a *checked type conversion* indicating that `expr_atom` shall be treated as the required type `ty`. See 7.13.1 *Checked type conversions on expressions*.

R$_{HDDS}$ The syntax

```
cexpr as <constraint>
```

is syntactic sugar for

```
cexpr as integer <constraint>
```

I$_{SBFK}$ Note that the checked type conversion operator binds at lowest precedence, so

```
M+N as K
```

is equivalent to

```
(M+N)as K.
```

Parentheses can be used when necessary, e.g.

```
M * (N + (S as T))
```

or to improve readability. It is recommended to use parentheses whenever there is a risk that the meaning might not be clear to readers of the specification.

5.12 The UNKNOWN expression

$\mathrm{R}_{\mathrm{XBMN}}$ The type of the expression `UNKNOWN: ty` is `ty`.

$\mathrm{R}_{\mathrm{WLCH}}$ The expression `UNKNOWN: ty` evaluates to an arbitrary value in the domain of `ty`.

$\mathrm{I}_{\mathrm{GJHS}}$ There is no `UNKNOWN` value which can be held in a storage element.

Chapter 6
Statement syntax

Statements consist of:

- simple statements including declarations, assignments, procedure calls and returns.
- conditional and case statements.
- repetitive statements: for, while and repeat loops.
- exception handling statements.

6.1 Statements

```
stmt ::= annotation stmt
       | decl_stmt
       | lexpr "=" expr ";"
       | identifier "(" null_or_expr_list ")" ";"
       | "return" expr_opt ";"
       | "assert" expr ";"
       | "throw" expr_opt ";"
       | "pass" ";"
       | "if" expr "then" stmt_list elsif_list else_opt "end"
       | "case" expr "of" alt_list otherwise_opt "end"
       | "for" identifier "=" expr direction expr "do" stmt_list "end"
       | "while" expr "do" stmt_list "end"
       | "repeat" stmt_list "until" expr ";"
       | "try" stmt_list "catch" catcher_list otherwise_opt "end"
       | "pragma" identifier null_or_expr_list ";"

stmt_list ::= stmt stmt_list
            | stmt
```

R_{PTNG} Statements consist of:

- Declarations of variables, let values and constants. See 6.2 *Local storage elements*.
- 6.5 *Assignment statements*.
- 6.3 *Procedure invocation statements*.
- 6.4 *Return statements*.
- 6.6 *Assertion statements*.
- Throw statements which throw exceptions. See 6.10.2 *Throw Statements*.
- Pass statements which do nothing (nop).
- 6.7 *Conditional statements*.
- 6.8 *Case Statements*.
- 6.9 *Repetitive statements*.
- 6.10 *Exception handling*.

6.2 Local storage elements

```
decl_stmt ::= "var" identifier ":" ty ";"
            | "var" identifier "," identifier_list ":" ty ";"
            | "var" decl_item "=" expr ";"
            | "let" decl_item "=" expr ";"
            | "constant" decl_item "=" expr ";"

decl_item ::= identifier ty_opt
            | "(" decl_item_list ")" ty_opt
            | "[" decl_item_list "]" ty_opt
            | "-" ty_opt

decl_item_list ::= decl_item "," decl_item_list
                 | decl_item

ty_opt ::= ":" ty
         |
```

This provides a way to define local *variables*, *let* values and *constants*.

R$_{\text{XSDC}}$ The `var` keyword is used to declare *local variable identifiers* denoting local storage elements which are all of the following: non-compile-time-constant, execution-time, mutable.

R$_{\text{BFWL}}$ The `let` keyword is used to declare *local let identifiers* denoting local storage elements which are all of the following: non-compile-time-constant, execution-time, immutable.

R$_{\text{TFJZ}}$ The `constant` keyword is used to declare *local constant identifiers* denoting local storage elements which are all of the following: compile-time-constant, non-execution-time, immutable.

R$_{\text{SQJJ}}$ The type of a local storage element declared with the `constant` keyword must be a compile-time-constant type. See also section 8.4.2 *Compile-time-constant types*.

R$_{\text{QDQD}}$ A single local identifier or a list of local identifiers may be declared along with an initialization expression. Each identifier may have an optional type annotation.

R$_{\text{CLQJ}}$ Where a hyphen (`-`) is present instead of an identifier, the corresponding initialization value will be discarded after evaluating it.

R$_{\text{FMLK}}$ One or more variables, all of the same type, may be declared simultaneously with no initialization expression. In this case, a single type annotation must be included after the last variable name in the declaration.

R$_{\text{PNQJ}}$ A declaration using a parenthesized list of identifiers (or nested parenthesized lists) requires an initializer expression with the structure of a tuple. The type for each of the identifiers may be omitted and instead the type for each local storage element will be the type of the positionally paired element (or nested tuple) of the initializer tuple expression.

I$_{\text{YPXD}}$ The initialization defined in rule PNQJ is similar to the destructuring assignment described in 6.5.2 *Multi-assignment*.

R$_{\text{KKDF}}$ Multiple variables, all of a determined width bitvector type, may be declared simultaneously using a concatenation declaration. In a concatenation declaration each variable being declared must have either: a type annotation that defines the determined width bitvector or have no type annotation. If no annotation is specified then bits(1) is used. A concatenation declaration requires an initialization expression that is a determined width bitvector that is the same width as the sum of the widths of variables being declared.

I$_{\text{SRQF}}$ Nesting of concatenated variable names in a concatenation declaration is allowed but does not define or imply any organization of the variables; i.e. nesting of concatenated variable names is not equivalent to Nested bitfields. See also 3.8.1.1 *Nested bitfields*.

R$_{\text{NXSF}}$ If a locally declared identifier has an associated type in the `declaration_stmt`, then the identifier has that type.

R$_{\text{XHPB}}$ If a locally declared identifier does not have an associated type then it must have an associated initialization expression and the identifier has the type of that expression.

6.2.1 Scope of local declarations

I_{KBXM} ASL has static scoping. This means that the local identifiers of a function or procedure are not in scope within any subprogram which it executes.

See also 4.1 *Scope of global declarations*.

R_{JFRD} A local identifier declared with `var`, `let` or `constant` is in scope from the point immediately after its declaration until the end of the immediately enclosing block.

I_{TTQJ} Rule JFRD means that it is not possible for a specification to define a cycle among the declarations and initializations of locally declared identifiers. See also info LWQQ for how global declarations differ in this respect.

R_{LCFD} A local declaration shall not declare an identifier which is already in scope at the point of declaration.

6.2.2 Initialization of locals

R_{KSZM} A local storage element declared with `var` is initialized with the base value of its type if no initialization expression is given.

I_{FKJC} Writing ASL specifications which rely on the implicit initialization of local variables is strongly discouraged. It is recommended that tools which process ASL attempt to detect code which reads local variables before writing to them and report this as an error which can be downgraded to a warning by users.

R_{XYLP} Where the initialization expression in a `declaration_stmt` is a bitvector of determined width, if the initialization expression type satisfies the declared type, then the declaration creates a storage element whose determined width is the determined width of the initialization expression.

R_{ZXHP} The initialization expression in a `let` or `var` declaration is evaluated during execution-time at the point of declaration, hence any side effects occur at the point of declaration.

R_{TZRV} The initialization expression in a local `constant` declaration must be a compile-time-constant expression.

R_{NFKG} A local storage element declared with `constant` is initialized with the value of its initialization expression during compilation.

Examples

Declaration of a local variable with identifier `size` of type `integer` and initial value 256:

```
var size: integer = 256;
```

The type may be omitted from the declaration if it can be unambiguously inferred from the initializer expression:

```
var size = 256;
```

Multiple variables may be declared in a single declaration:

```
var x, y, z: integer;
```

The following declares three local variables which are non-compile-time-constant, execution-time and mutable:

- `myF` is an integer, initialized with the result of invoking `f()`
- `size` is initialized with the integer `256`
- `myH` is initialized with the result of invoking `h()`
- Since the type of `myH` is not specified, it is derived from the return type of `h()`
- The invocation of `g()` occurs but the result is discarded.

```
var (myF: integer, -, (size, myH)) = (f(), g(), (256, h()));
```

The following concatenation declaration defines five local `let` variables. It uses a concatenation assignment that requires a bitvector on the right-hand side that has a total width equal to the sum of the declared widths of the variables being defined.

```
    let [ v1 : bits(4), [ v2 : bits(8), v3 : bits(16) ], v4, v5 : bits(3) ] =
        [ vec8_a, vec16_a, vec8_b ];

    // the above single let declaration is equivalent to the combinaton of
    // the five declarations:
    // let v1 : bits(4) =  vec8_a[7:4];
    // let v2 : bits(8) = [ vec8_a[3:0], vec16_a[15:12] ];
    // let v3 : bits(16) = [ vec16_a[11:0], vec8_b[7:4] ];
    // let v4 : bits(1) = vec8_b[3];  // v4 has unspecified type -- bits(1) is used.
    // let v5 : bits(3) = vec8_b[2:0];
```

6.3 Procedure invocation statements

```
stmt:invoke ::= identifier "(" null_or_expr_list ")" ";"
```

D_{KCYT} A procedure invocation statement calls a procedure subprogram using the given actual arguments. The subprogram must not have a return type.

For more details on how procedure invocations are resolved, see 7.8.2 *Subprogram invocations*.

6.4 Return statements

```
stmt:return ::= "return" expr_opt ";"

expr_opt ::= expr
           |
```

D_{HTPL} A return statement returns the control flow to the caller of a subprogram.

R_{PHNZ} A return statement appearing in a getter or function requires a return value expression that type-satisfies the return type of the subprogram.

R_{NYWH} A return statement appearing in a setter or procedure must have no return value expression.

6.5 Assignment statements

```
stmt:assign ::= lexpr "=" expr ";"

lexpr ::= "-"
        | lexpr_atom
        | "(" lexpr_list ")"

lexpr_atom ::= identifier
             | lexpr_atom "." identifier
             | lexpr_atom "." "[" identifier_list "]"
             | "[" lexpr_atom_list "]"
             | lexpr_atom "[" null_or_slice_list "]"
```

D_{BJNY} An assignment statement stores the value of the right-hand side expression in one or more storage elements, as denoted by the left hand side.

6.5.1 Left hand side rules

The left hand side of an assignment statement can contain simple pattern matches consisting of:

- '-' (hyphen, minus sign) meaning discard the value
- An in-scope, mutable identifier. This can be one of:
  - A local variable.
  - A global variable.
  - A call to an unparameterized setter function.

  These are disambiguated based on whether there is a global variable or unparameterized setter function in scope.
- Field assignment: denoted `lexpr_atom.identifier` where `lexpr_atom` is an expression that yields a record, exception or bitvector value, and `identifier` is the field identifier. Fields can be fields of records, exceptions, or bitfields of bitvectors (see 3.8.1 *Bitfields*).
- Multiple bitfield assignment: each bitfield is assigned the corresponding slice of the right-hand side bitvector expression. For example, if the bitvector `P` has a 3-bit field `X` and a 2-bit field `Y`, then `P.[X,Y] = '101 00';` assigns `P.X` the value `'101'` and assigns `P.Y` the value `'00'`.

R_{ZHYT} It is an error if any field in a multiple field assignment is not of type bitvector.

R_{YYFR} The fields in a multiple field assignment must not overlap.

- An indexed write. This can be one of
  - Assignment to an element of an array.
  - Assignment to a bitslice.
  - A call to a parameterized setter function.

    These are disambiguated based on whether there is a global variable or parameterized setter function in scope and on whether the type of the variable is a bitvector or an array.
- Multiple values can be assigned at once using a multi-assignment statement. See 6.5.2 *Multi-assignment*.
- Multiple bitvector values can be assigned at once using concatenation. Each bitvector is assigned the corresponding bitslice of the right-hand side expression.

Examples:

```
    size = 32; //  assign to variable
```

```
    PSTATE.nRW = TRUE; //  write to `nRW` field of the `PSTATE` variable
```

```
PSTATE.[N,Z,C,V] = '0000'; //  Using multiple bitfield assignment, write to multiple
    ↪fields of the `PSTATE` variable
```

```
EPSR[26:25, 15:10] = Zeros(8); //  write to bitslice of `EPSR` variable
```

```
[n, z, c, v] = '0000'; //  Using concatenation assignment, bitslice the literal constant
    ↪and assign each slice to individual variables
```

6.5.2 Multi-assignment

D_{WSXY}

D$_{\texttt{WSXY}}$ Multi-assignment is a destructuring assignment that allows the elements of a tuple on the right-hand side of an assignment to be stored in different storage elements as denoted by the items in the parenthesized list on the left-hand side.

R$_{\texttt{ZHVH}}$ The number of elements in the tuple on the right-hand side of the assignment must be equal to the number of items (name or dash) in the parenthesized list on the left-hand side.

I$_{\texttt{CJVD}}$ The items (name or dash) in the parenthesized list on the left-hand side are positionally paired with the elements from the tuple on the right-hand side. For each pair, an assignment is made from the right-hand side element to the left-hand side element.

I$_{\texttt{SSXJ}}$ A dash indicates that the element at that position in the tuple on the right-hand side is not assigned to anything, and is instead discarded.

Examples:

```
(a, b) = (1, 2); // equivalent to a = 1; b = 2;
```

```
// Discard the second element returned by AddWithCarry
(result, -) = AddWithCarry(R[n], R[m], '0');
```

6.5.3 Setter invocation

R$_{\texttt{YDFQ}}$ Where the left hand side of an assignment is to an identifier which is declared as a setter, the assignment is treated in the same way as a procedure invocation. In this case, if a sequence of `null_or_slice_list`s is present, each must consist of a single expression. The sequence of `null_or_slice_list`s shall be used as the actual expressions for the invocation of the setter. (see 7.8.2 *Subprogram invocations*)

Examples:

```
DHCSR = Zeros(32); //  call unparameterized setter function `DHCSR`
```

```
Mem[addr, 32] = Zeros(32); //  call parameterized setter function `Mem`
```

6.6 Assertion statements

D_{QJYV} An assertion statement takes an expression that is asserted by the specification to be TRUE when the assertion statement is executed.

R_{WZSL} If an assertion expression is FALSE when the assertion statement is executed, it indicates an error in the specification. An implementation may throw an implementation-defined exception in this case, as described in 10.1 *Dynamic errors*, but is not required to.

R_{WQRN} The expression in an assertion statement must be side-effect-free.

6.7 Conditional statements

```
stmt:if ::= "if" expr "then" stmt_list elsif_list else_opt "end"

else_opt ::= "else" stmt_list
           |

elsif ::= "elsif" expr "then" stmt_list
```

R_TMYS Conditional statements select which block to execute by testing condition expressions sequentially until a `TRUE` condition is found.

R_XSSL If no `TRUE` condition is found and there is an `else` block, the `else` block is executed.

R_KZTJ If no `TRUE` condition is found and there is no `else` block, no block is executed.

Examples:

```
    if x > y then
        return 1;
    elsif x < y then
        return -1;
    else
        return 0;
    end
```

```
    if d IN {13,15} || n IN {13,15} then
        UNPREDICTABLE();
    end
```

```
    if size == '01' then
        esize = 16; elements = 4;
    end
```

6.8 Case Statements

```
stmt:case ::= "case" expr "of" alt_list otherwise_opt "end"

alt_list ::= alt alt_list
           |

alt ::= "when" pattern_list where_opt "=>" stmt_list

where_opt ::= "where" expr
            |

otherwise_opt ::= "otherwise" "=>" stmt_list
                |
```

$\mathrm{I}_{\mathrm{CHGZ}}$ The expression following the `case` keyword is called the discriminant.

$\mathrm{R}_{\mathrm{PSZY}}$ The discriminant is evaluated once each time the case statement is evaluated.

$\mathrm{R}_{\mathrm{JHST}}$ Case alternatives are examined in sequential order. If any of the patterns match the discriminant (and the guard expression is true, if present - see below) then this case alternative is considered selected, its statement list is executed, and the case statement ends without examining any further case alternatives.

$\mathrm{R}_{\mathrm{RXQB}}$ Testing the discriminant against a pattern list follows the same procedure described in 5.3 *Pattern matching*. It is not an error if it can be statically determined that none of the patterns in a case alternative can match the discriminant.

$\mathrm{R}_{\mathrm{ZYVW}}$ If no case alternative is selected, and there is an `otherwise_opt` block, the `otherwise_opt` block is executed.

$\mathrm{R}_{\mathrm{JVTR}}$ If no case alternative is selected, and there is no `otherwise_opt` block, it indicates an error in the specification, and an implementation defined exception may be thrown.

Examples:

```
    case spreg of
        when RNamesSP_Main_Secure       => limit = [MSPLIM_S.LIMIT, '000'];
        when RNamesSP_Process_Secure    => limit = [PSPLIM_S.LIMIT, '000'];
        when RNamesSP_Main_NonSecure    =>
            limit = if HaveMainExt() then [MSPLIM_NS.LIMIT, '000'] else Zeros(32);
        when RNamesSP_Process_NonSecure =>
            limit = if HaveMainExt() then [PSPLIM_NS.LIMIT, '000'] else Zeros(32);
            assert (FALSE);
    end
```

```
    case size of
        when '01' => S[d] = [Zeros(16), FPAbs(S[m][15:0])];
        when '10' => S[d] = FPAbs(S[m]);
        when '11' => D[d] = FPAbs(D[m]);
        // note that it would be an error if size == '00'
    end
```

```
    // This example is not valid ASL
    case b of
        when '10' =>   // empty statement list is invalid, does not fall through
        when '11' =>
            X[30] = 0;
    end
```

6.8.1 Case guards

$\mathrm{R}_{\mathrm{CWNT}}$ A case alternative in a case statement may be optionally guarded with a condition expression, indicated by the inclusion of the `where` keyword. Only if the pattern match is successful is the guard expression evaluated. The guard expression must evaluate to `TRUE` for the case alternative to be selected.

Example:

```
// in the following 'd' is evaluated once,
// if d == '01' and a == 5 then the sub-expression 'a' will be evaluated twice
case d of
    when '00'              => return 1;

    // if d matches '01' then evaluate (a>8), if true then return 9
    when '01' where a > 8  => return 9;

    // if d matches '01' then evaluate (a<3), if true then return 2
    when '01' where a < 3  => return 2;
    when '01'              => return 3;
    when '10'              => return 4;
    when '11'              => return 5;
end
```

6.9 Repetitive statements

```
stmt:for ::= "for" identifier "=" expr direction expr "do" stmt_list "end"

direction ::= "to"
            | "downto"

stmt:while ::= "while" expr "do" stmt_list "end"

stmt:repeat ::= "repeat" stmt_list "until" expr ";"
```

R$_{SSBD}$ For statements introduce a *for-loop counter* which is in scope for the body of the for statement.

R$_{ZSND}$ If either the start or end expression of the for-statement are unconstrained integers then the for-loop counter is an unconstrained integer. If either the start or end expression of the for-statement are under-constrained integers and neither the start nor the end expression are unconstrained integers then the for-loop counter is an under-constrained integer. Otherwise the for-loop counter is a constrained integer whose constraint is:

- `min(SC)..max(EC)U SC` when the `direction` is `to` and `min(SC)<= max(EC)`
- `SC` when the `direction` is `to` and `min(SC)> max(EC)`
- `max(SC)..min(EC)U SC` when the `direction` is `downto` and `max(SC)>= min(EC)`.
- `SC` when the `direction` is `downto` and `max(SC)< min(EC)`.

where `SC` denotes the domain of the type of the start expression, `EC` denotes the domain of the type of the end expression, `x..y` denotes the closed integer interval between `x` and `y` inclusive and `U` denotes the set union operation

R$_{RQNG}$ For-loop counters are immutable, execution-time storage elements.

R$_{YTNR}$ The start expression shall evaluate to a value that remains unchanged if the start expression were to be re-evaluated at the beginning of each for-loop iteration.

R$_{NZGH}$ The end expression shall evaluate to a value that remains unchanged if the end expression were to be re-evaluated at the beginning of each for-loop iteration.

R$_{KLDR}$ The start expression and the end expression must not be side-effecting.

R$_{LSVV}$ The for statement is executed by initializing the for-loop counter with the value of the start expression, and then repeating the following actions until the for statement is completed:

- if the `direction` is `to` (`downto`) and the loop counter value is greater than (less than) the end expression then the for statement is considered completed
- the body of the for statement (`stmt_list`) is executed
- the value of the for-loop counter is updated by incrementing (decrementing) it by 1 if the `direction` is `to` (`downto`)

I$_{KFYG}$ ASL does not provide a break statement. If early termination of a for-loop is required, consider rewriting it using either the `while` or `until` loop construct.

I$_{YDBR}$ If the start expression is greater than (less than) the end expression in a `to` (`downto`) for-loop, the `stmt_list` of the for-loop is not executed.

Examples:

```
    for i = 0 to 12 do
        R[i] = Zeros(32);    // this line is executed 13 times to set R[0] ... R[12]
    end

    for i = 12 to 0 do
        S[i] = Zeros(32);    // this line is never executed
                             // the initial value (12) is greater than the end expression (0)
    end
```

R$_{MHPW}$ While statements repeatedly test a condition expression and then execute a block of statements.

R$_{\text{JQXC}}$ The while loop terminates (without executing the block again) once the condition is `FALSE`.

Examples:

```
    @looplimit(32)
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;
        exponent = exponent - 1;
    end
```

R$_{\text{NPWR}}$ Repeat statements repeatedly execute a block of statements and then test a condition expression.

R$_{\text{WVQT}}$ The repeat loop terminates once the condition is `TRUE`.

Examples:

```
    @looplimit(20)
    repeat
        emptySlot = !_InstInfo[i].Valid;
        if emptySlot && (isBeatInst || i == 0) then
            _InstInfo[i].Valid  = TRUE;
            _InstInfo[i].Length = len;
            _InstInfo[i].Opcode = opcode;
        end
        i = i + 1;
    until emptySlot || (!isBeatInst && i > 0) || (i >= MAX_OVERLAPPING_INSTRS);
```

6.9.1 Loop limits

R$_{\text{XWRB}}$ Repetitive statements must provide a finite upper bound on the maximum number of iterations the statement can perform. The upper bound of a for-statement can be inferred from the range of its for-loop counter variable if it is a constrained integer. Otherwise the statement must be annotated with an upper bound via a `@looplimit` annotation.

R$_{\text{CQSX}}$ While and repeat statements always require `@looplimit` annotations.

R$_{\text{PMKP}}$ The loop limit must be a constrained integer expression.

R$_{\text{LPVP}}$ When the limit is exceeded, an error should occur.

Examples:

```
    for i = 0 to 12 do
        R[i] = Zeros(32);   // this line is executed 13 times to set R[0] ... R[12]
    end
```

The upper bound of iterations can be determined by the range of `i` which is `0..12`.

```
    let N: integer{10..20} = ...;
    for i = 0 to N do
        R[i] = Zeros(32);   // this line is executed up to 20 times to set R[0] ... R[19]
    end
```

The upper bound of iterations can be determined by the range of `i` which is `0..N`. Since the range of `N` is `10..20`, the range of `i` must therefore be `0..20`.

```
    @looplimit(20)
    while not_completed() do
        i = i + 1;
    end
```

The above loop will perform a maximum of 20 iterations, and on the 21st iteration a runtime error shall occur.

6.10 Exception handling

```
stmt:try ::= "try" stmt_list "catch" catcher_list otherwise_opt "end"

catcher ::= "when" identifier ":" ty "=>" stmt_list
          | "when" ty "=>" stmt_list

catcher_list ::= catcher catcher_list
               |
```

6.10.1 Try statements

```
stmt:try ::= "try" stmt_list "catch" catcher_list otherwise_opt "end"

otherwise_opt ::= "otherwise" "=>" stmt_list
                |
```

R_{DGRV} Execution of a `try` statement proceeds by executing its `stmt_list`. If execution reaches the end of the `stmt_list` then execution proceeds to the statement following the `try` statement.

6.10.2 Throw Statements

See 6.1 *Statements* for syntax.

R$_{\text{TXTC}}$ The `throw` statement causes the exception value obtained by evaluating its expression to be thrown. The *thrown type* of the thrown value is the type of the expression.

R$_{\text{GVCC}}$ When an exception is thrown, execution proceeds to the `catch` of the innermost `try` of the enclosing subprogram in the call stack. If there is no enclosing `try` statement when an exception is thrown then execution terminates.

6.10.3 Catchers

```
catcher ::= "when" identifier ":" ty "=>" stmt_list
          | "when" ty "=>" stmt_list

catcher_list ::= catcher catcher_list
               |

otherwise_opt ::= "otherwise" "=>" stmt_list
                |
```

R$_{\text{SPNM}}$ When the `catch` of a `try` statement is executed, then the thrown exception is caught by the first `catcher` in that `catch` which it type-satisfies or the `otherwise_opt` in that `catch` if it exists.

R$_{\text{ZTLB}}$ If a thrown exception is not caught in a `catch` then the exception is passed to the `catch` of the closest enclosing `try` statement in the current execution.

R$_{\text{YVXF}}$ When an exception is caught by a `catcher` or an `otherwise_opt`, the `stmt_list` corresponding to that `catcher` or the `otherwise_opt` is executed respectively.

D$_{\text{GGCQ}}$ The `catcher` syntax includes an optional `identifier` immediately following the `when` keyword. That identifier is called the catcher's exception.

D$_{\text{JTDG}}$ The *caught type* of a catcher's exception is the type it is annotated with in the `catcher`.

R$_{\text{DHKH}}$ When a catcher's `stmt_list` begins executing, the catcher's exception is declared (if provided) and denotes a local, immutable, execution-time storage element which is initialised with the value of the exception which the catcher caught.

R$_{\text{MKGB}}$ A catcher's exception is only in scope within the `stmt_list` of the `catcher`.

I$_{\text{GZVM}}$ The catcher's exception may have a *caught type* which is not the same as the *thrown type* of the original exception which was caught.

R_{JZST} If execution reaches the end of a `stmt_list` corresponding to a `catcher` or an `otherwise_opt` then execution proceeds to the statement following the `try` statement which contains the `catcher` or the `otherwise_opt`.

I_{YWKG} It is implementation defined whether or not a dynamic error (such as an assertion failure, divide by zero, index out of bounds etc.) can be caught as an exception. See 10.1 *Dynamic errors*.

Examples:

```
    try
        could_throw_exception();
    catch
        when Excp =>
            handle_excp();
            throw; // this will rethrow the handled exception
        when exn: Excp2 =>
            handle_excp2(exn);
            throw exn; // throws exn, this will have the same result with throw;
        otherwise =>
            unhandled();
    end
```

6.10.4 Rethrowing exceptions

R_{BRCJ} An expressionless `throw` statement may only be used in the `stmt_list` of a `catcher`.

R_{GVKS} An expressionless `throw` statement causes the exception which the currently executing `catcher` caught to be thrown.

I_{YKLF} The exception thrown by an expressionless `throw` statement has the same *thrown type* of the original exception which the current catcher caught and may not be the same as the *caught type* of the catcher's exception.

Examples:

```
type BAD_OPCODE of exception;

func decode_instruction(op: bits(32))
begin
    if op[1] == '1' then
        throw BAD_OPCODE;
    end
end

func top(opcode: bits(32))
begin
    try
        decode_instruction(opcode);
    catch
        when e: BAD_OPCODE =>
            handle_exception();
    end
end
```

In the above example, the exception `BAD_OPCODE` thrown in `decode_instruction` is caught in the calling subprogram `top` where the catcher proceeds to call `handle_exception`.

```
type IsExceptionTaken of exception;

func HandleException()
begin
    try
        if HandleExceptionTransitions(commitState) then
            SteppingDebug();
            VectorCatchDebug();
        end
        if LockedUp && NextInstrAddr() != 0xEFFFFFFE[31:0] then
            LockedUp = FALSE;
        end
        if !LockedUp then
```

```
            InstructionAdvance(commitState);
        end
    catch
        when exn: IsExceptionTaken =>
            pass; // ignore exception
    end
end
```

6.11 Pragmas

Listing 6.1: Definition of pragma

```
stmt:pragma ::= "pragma" identifier null_or_expr_list ";"
```

I_{FVRF} Pragmas can be used at the declarative scope see 4.6 *Pragmas* and within a subprogram as a statement.

```
func my_function_with_tool_pragmas()
begin
    pragma my_tool_pragma1;
    pass;
end
```

Chapter 7
Type inference and type-checking

ASL statically checks that expressions are correctly typed. In this section we describe the rules for type checking.

This section is an informal description of how ASL is interpreted. It is expected to be replaced by a Semantics Reference Document in due course. If there is a conflict between the behavior described in this document and the Semantics Reference Document, the Semantics Reference Document will be considered the correct version.

I_{LDNP} See also Chapter 3 *Builtin Types* for details on type declarations.

7.1 Type nomenclature

In this section we define some basic terminology about types.

7.1.1 Named, Anonymous and Primitive types

D$_{\texttt{VMZX}}$ All types are either:

- *named* types: those which are declared using the `type` syntax.
- or *anonymous* types: those which are not declared using the `type` syntax.

D$_{\texttt{GWXK}}$ All types are either:

- *primitive* types: those which only uses the builtin types (see rule NZWT)
- or *non-primitive* types: those which are named types or which make use of named types.

7.1.2 Structure of a type

D$_{\texttt{FXQV}}$ The *structure* of a type is the primitive type it is equivalent to such that it can hold the same values.

Examples:

```
type T1 of integer;         // the named type `T1` whose structure is integer
type T2 of (integer, T1);   // the named type `T2` whose structure is (integer, integer)
// Note that (integer, T1) is non-primitive since it uses T1

var x: T1;
// the type of x is the named (hence non-primitive) type `T1`
// whose structure is `integer`

var y: integer;
// the type of y is the anonymous primitive type `integer`

var z: (integer, T1);
// the type of z is the anonymous non-primitive type `(integer, T1)`
// whose structure is `(integer, integer)`
```

7.1.3 Domain of Values for Types

D$_{\texttt{BMGM}}$ The *domain* of a type is the set of values which storage elements of that type may hold.

Examples:

- The domain of `integer` is the infinite set of all integers.
- The domain of `bits(1)` is the set {‘1’, ‘0’}
- The domain of `integer {2,16}` is the set containing the integers 2 and 16.
- The domain of `bits({2,16})` is the set containing all two bit and all sixteen bit values.

7.2 Execution-time checks

R_{WZVX} An execution-time check is a condition that is evaluated during the evaluation of an execution-time initializer expression or subprogram. If the condition evaluates to `FALSE` a dynamic error is generated (see 10.1 *Dynamic errors*).

R_{VBLL} Wherever an execution-time check is required, a tool may elide the check if it can be proven at compile time to always be true.

I_{KRLL} If a compiler can prove that an execution-time check is always false then it may issue a warning. Note that it may be an error if the compiler can also prove that the execution time check will always be executed.

7.3 Constrained types

ASL allows the use of *bitvectors whose width is non-compile-time-constant*.

Where a bitvector has variable width, it is desirable to indicate the set of possible widths in the code so that pseudocode readers are not left guessing which widths are architecturally valid.

To this end we introduce a type of integer whose value is limited to a finite set of integer values. We refer to these as *constrained integers*. Note that this is a constraint on the values of the integer typed object, it is not a constraint on the types which the object may have. The object's type is still `integer`, but its *domain* is restricted.

We also introduce *checked type conversions* which are an annotation on an expression indicating the type the expression is believed to be able to have (according to the author of the code…)

We then require that bitvector *storage element's* widths are *constrained integers*.

- For compile-time-constant widths, this is trivially satisfied by implicit constraints as described in 7.7.5 *Implicit constraints for compile-time-constant integer expressions*.
- For execution-time widths, we expect the use of *checked type conversions* on the width expression to suffice.

Various syntactic sugar is given so as to reduce clutter caused by *checked type conversions*.

It is also desirable that functions can be understood by readers in isolation which means *constraints* on bitvector widths may also be given in the function prototype.

The type checking rules for constrained types are designed to be usable on a function by function basis in order to allow local reasoning for authors and readers about the type correctness of a function in isolation.

$\mathrm{G}_{\mathrm{PFRQ}}$ Functions which are defined over arbitrary width bitvectors currently require a separate implementation in some translations for every width that can be used by the pseudocode. If those widths are enumerated as a constraint whose domain consists solely of compile-time-constant expressions, then tools may compute the possible widths.

Note that it is not a requirement for subprogram declarations to include *checked type conversions*, since the set of widths used by the pseudocode is actually determined from the invocations of the function. For example, the set of invocations of `f` in the following complete specification is constrained to the case where `N IN {4,8,16}`

```
func f{N}(x: bits(N)) => bits(N+1)
begin
    return [x, '0'];
end

func main()
begin
    var inputA: bits(4);
    var outputA = f(inputA); // an invocation of f: bits(4)=>bits(5)

    let widthB: integer {8,16} = if (cond) then 8 else 16;
    var inputB: bits(widthB);
    var outputB = f(inputB); // an invocation of f: bits({8,16})=>bits({9,17})
    // outputB is of type bits({9,17})
end
```

Note that in the following example, the set of invocations of `append` can also be determined without evaluating any non-compile-time-constant expressions. See also 7.8.5 *Primitive operations on integers*.

```
func append{N,M}(x: bits(N), y: bits(M)) => bits(N+M)
begin
    return [x, y];
end

func main()
begin
    var inputA: bits(4);
    var outputA = append(inputA, inputA);
    // an invocation of append: (bits(4), bits(4))=>bits(8)
    let width1: integer {8,16} = if (cond1) then 8 else 16;
    var input1: bits(width1);
    let width2: integer {8,16} = if (cond2) then 8 else 16;
    var input2: bits(width2);

    var output12 = append(input1, input2);
    // an invocation of append: (bits({8,16}), bits({8,16}))=>bits({16,24,32}))
    // Compiler can enumerate all possible invocations:
    // append: (bits( 8), bits( 8))=>bits(16)
    // append: (bits( 8), bits(16))=>bits(24)
    // append: (bits(16), bits( 8))=>bits(24)
    // append: (bits(16), bits(16))=>bits(32)
    // Hence output12 is bits({16,24,32})
end
```

7.4 Constrained Integers

$\mathrm{D_{ZXSS}}$ Integer types are either *unconstrained* or *constrained*.

$\mathrm{D_{ZTPP}}$ Constrained integer types are either *well-constrained* or *under-constrained*.

$\mathrm{R_{WJYH}}$ The integer type with no constraint is called the *unconstrained integer*.

$\mathrm{R_{HJPN}}$ An integer type with a constraint is called a *constrained integer*.

$\mathrm{R_{CZTX}}$ An integer type with a non-empty constraint is called a *well-constrained integer*.

$\mathrm{R_{TPHR}}$ The integer type with an empty constraint is called the *under-constrained integer*.

$\mathrm{I_{ZDDJ}}$ There is no syntax for an integer with an empty constraint, although parameters of subprograms can declare under-constrined integers: see 7.8.1 *Dependently typed bit vector formals*.

$\mathrm{I_{GHGK}}$ The under-constrained integer is used as the implicit type of unconstrained integer subprogram parameters. Intuitively it indicates that the parameter has a *single value provided by the subprogram's invocation* but the range of possible values is not known when the subprogram declaration is type checked. This allows parameters to be used as bitvector widths in a subprogram when their constraint is not given in the subprogram declaration, without causing type checking errors. See also 7.8.1 *Dependently typed bit vector formals* and 7.8.5 *Primitive operations on integers*.

$\mathrm{R_{LSNP}}$ A constraint is specified as a list of constraint ranges, where each range consists of a single statically evaluable, constrained integer expression or a lower and upper bound range (inclusive) using the syntax `expr .. expr` where the bounds are statically evaluable, constrained integer expressions.

$\mathrm{R_{BSMK}}$ The values in a constrained integer's constraint must all be statically evaluable, constrained integer expressions.

$\mathrm{R_{PHRL}}$ A constrained integer's constraint must be a finite set.

7.4.1 Domain of integers

$\mathrm{R_{TZNR}}$ The domain of a constrained integer type with a constraint that contains at least one constraint range that contains an under-constrained integer expression is the domain of the under-constrained integer

$\mathrm{R_{RLQP}}$ The domain of a constrained integer type with a constraint whose constraint ranges contain only well-constrained integer expressions is the union of the domain of its constraint ranges.

$\mathrm{R_{LYDS}}$ The domain `D` of a constraint range that contains well-constrained integer expressions is recursively defined as such:

- If the constraint range consists of a single well-constrained integer expression that is a compile time constant `c`, `D = {c}`
- If the constraint range consists of a lower and upper bound range `c..d` where `c` and `d` are well-constrained integer expressions that are compile time constants then:
  - if `c` <= `d` then `D = [c..d]`
  - otherwise the constraint range is illegal and should generate a type-checking error
- If the constraint range consists of a single statically evaluable well-constrained integer expression `E`, `D` is equal to the domain of the type of `E`
- Consider a constraint range that consists of a lower and upper bound range `Ec..Ed` where both `Ec` and `Ed` are statically evaluable, well-constrained integer expressions. Let `LD` and `UD` be the domain of the type of `Ec` and `Ed` respectively, where `mLD`/`MLD` denotes the minimum/maximum element of `LD` and `mUD`/`MUD` denotes the minimum/maximum element of `UD`.
  - if `MLD <= mUD` then `D = [mLD..MUD]`
  - otherwise the constraint range is illegal and should generate a type-checking error.

$\mathrm{R_{SVDJ}}$ The domain of every well-constrained integer type is a proper subset of the domain of the under-constrained integer type.

$\mathrm{I}_{\mathrm{WLPJ}}$ Hence the domain of the under-constrained integer type is not a subset of the domain of any well-constrained integer type.

$\mathrm{R}_{\mathrm{FWMM}}$ The domain of the under-constrained integer type is a proper subset of the domain of the unconstrained integer type.

$\mathrm{I}_{\mathrm{WBWL}}$ Hence the domain of the unconstrained integer type is not a subset of the domain of the under-constrained integer type.

$\mathrm{I}_{\mathrm{CDVY}}$ It follows from the domain rules that well-constrained integers type satisfy under-constrained integers, so a well-constrained integer may be used wherever an under-constrained integer is required by uses of the type-satisfaction rule, for example, a well-constrained integer may be an actual argument when the formal argument is an under-constrained integer.

$\mathrm{I}_{\mathrm{KFCR}}$ It follows from the domain rules that under-constrained integers type-satisfy unconstrained integers but not vice versa so an unconstrained integer may not be used where a constrained integer is required by uses of the type-satisfaction rule, even if the constrained integer is under-constrained.

$\mathrm{I}_{\mathrm{BBQR}}$ Domains for integers are related in the following way:

well-constrained integer $\subset$ under-constrained integer $\subset$ unconstrained integer

7.5 Constraints on bitvector widths

I_{SCBX} Bitvectors can never have a width which is unconstrained.

7.5.1 Types of bitvector

I_{MRHK} Unlike integers which may be unconstrained and hence have an infinite domain, bitvectors must always have a finite domain. Hence all bitvectors are "constrained" in the sense that their domain is not infinite. However, we use the term "constrained width" bitvector to indicate that the type has a range of possible widths and we will refer to bitvectors which only have one *possible* width as "fixed width" bitvectors.

D_{NRWC} Bitvectors are either *constrained width bitvectors* or *fixed width bitvectors*.

D_{WSZM} Bitvectors are either of *undetermined width* or have a *determined width*.

D_{BVGK} A *constrained width bitvector* whose width is given by an expression of type `ty` that has the structure of an under-constrained integer is an *under-constrained width bitvector*. Under-constrained width bitvectors have a *determined width*.

D_{CBQK} Fixed width bitvectors have a *determined width*.

7.5.2 Bitvectors of the form `bits(-: ty)`

R_{QYZD} If `ty` has the structure of the unconstrained integer then `bits(-: ty)` is illegal.

R_{NFBN} If `ty` has the structure of a well-constrained integer whose domain contains only one value then `bits(-: ty)` denotes a *fixed width bitvector* whose *determined width* is equal to the value in the domain of `ty`.

R_{LJBG} If `ty` has the structure of a well-constrained integer whose domain contains more than one value then `bits(-: ty)` denotes a *constrained width bitvector* of *undetermined width*.

I_{YBHF} Note that the *under-constrained width bitvector* of *undetermined width* cannot be represented since there is no concrete syntax `ty` with the structure of the under-constrained integer.

R_{FZSD} The syntax

```
bits( <constraint> )
```

is syntactic sugar for

```
bits( -: integer <constraint> )
```

Note

We could instead add `ty : = constraint` as sugar for `integer constraint`?

7.5.3 Bitvectors of the form `bits(expr)`

R_{GHRP} If the type of `expr` has the structure of the unconstrained integer then `bits(expr)` is illegal.

R_{QZJS} If `expr` is of type `ty` which has the structure of a well-constrained integer whose domain contains only one value then `bits(expr)` denotes a *fixed width bitvector* whose *determined width* is equal to the value in the domain of `ty`.

I_{JSKW} Note that type checking rules mean that if `expr` is of type `ty` which has the structure of a well-constrained integer whose domain contains only one value, then `expr` evaluates to that value.

R_{SKRK} If `expr` is of type `ty` which has the structure of a well-constrained integer whose domain contains more than one value then `bits(expr)` denotes a *constrained width bitvector* whose *determined width* is `expr` and whose width is constrained as per the constraint on `expr`.

R$_{\text{NCNQ}}$ If `expr` is of type `ty` which has the structure of the under-constrained integer then `bits(expr)` is an *under-constrained width bitvector* and its *determined width* is `expr`.

I$_{\text{GQMV}}$ If `expr` is a compile-time-constant expression whose type is a well-constrained integer whose domain contains multiple values, then even though the width of the bitvector is a known integer value, the type of the bitvector is still a constrained width bitvector and not a fixed-width bitvector. For example, `bits(4 as integer {4,8})` is a constrained width bitvector whose determined width is `4`. (It is not clear that there is any practical use for this, but it is a corollary of other rules.)

7.5.4 Summary of types of bitvector

- Summary of valid types of bitvectors
  - constrained width bitvectors
    - ∗ of undetermined width
      - · `bits({32,64})`
    - ∗ of determined width
      - · `bits(expr)` (where `expr` is a statically evaluable expression of type `integer {32,64}`) is of determined width `expr` and its width is constrained to be `IN {32,64}`
    - ∗ under-constrained width bitvectors of undetermined width: not possible
    - ∗ under-constrained width bitvectors of determined width
      - · `bits(N)` where N is an under-constrained integer parameter
  - fixed width bitvectors
    - ∗ of undetermined width: not possible
    - ∗ of determined width:
      - · `bits({8})`
      - · `bits(k)` where `k` is of type `integer{8}`
- Summary of invalid types of bitvectors
  - unconstrained width bitvectors
    - ∗ `bits(-: integer)`
    - ∗ `bits(i)` where `i` is an unconstrained integer

7.5.5 Domain of a bitvector

R$_{\text{ZRWH}}$ Each bit within a bitvector has value ‘0’ or ‘1’.

I$_{\text{VMKF}}$ A bitvector can be compared for inclusion against a `bitmask_lit` as described in 5.3 *Pattern matching*.

R$_{\text{QXGW}}$ The type `bit` is identical to the type `bits(1)`

I$_{\text{RXLG}}$ As well as supporting bitwise logical operators and equality tests, bitvectors also support addition and subtraction, but are treated as not having a sign and therefore do not support ordering comparisons such as `<=`.

R$_{\text{ZWGH}}$ The domain of a fixed width bitvector is the set of values which can be represented by bitvector literals which are of the same length as the fixed width bitvector and which consist of the characters ‘0’ and ‘1’.

I$_{\text{SQVV}}$ For example, the bitvector `'1101'` has type `bits(4)`. The bitvector `''` has type `bits(0)`.

R$_{\text{PMQB}}$ The domain of a constrained width bitvector is the union of the domains of all bitvectors whose width is equal to a member of the bitvector’s width constraint.

R$_{\text{DKGQ}}$ The domain of every bitvector type other than the under-constrained width bitvector, is not a subset of the domain of the under-constrained width bitvector.

I$_{\text{MPSW}}$ In practice, this forbids the *assignment* of anything other than an under-constrained width bitvector of matching width to an under-constrained width bitvector. However, values which are not under-constrained width bitvectors may be used as actual arguments for an under-constrained width bitvector formal argument in an *invocation*, since type satisfaction is performed against the invocation type which is always an under-constrained width bitvector of determined width.

R$_{\text{DHZT}}$ The domain of the under-constrained width bitvector is not a subset of the domain of any other bitvector type

I_{VPST} In practice, this forbids the assignment of an under-constrained bitvector to anything other than an under-constrained bitvector of the same determined width.

7.5.6 Use of bitvectors of undetermined width

R$_{\text{VCZX}}$ A constrained width bitvector of undetermined width may only be used in the following places:

- As part of a checked type conversion.
- As part of the type of a storage element's declaration where an initialization expression is given.
- As part of the type of a formal argument in a subprogram declaration.

I$_{\text{TZVJ}}$ Note that a constrained width bitvector of undetermined width (e.g. `bits(-: integer{4, 8})`, `bits(integer{4, ↪ 8})`) may not be used as part of a return type. However, a constrained width bitvector of determined width (e.g. an under-constrained width bitvector `bits(N)` where `N` is an under-constrained integer, a constrained width bitvector `bits(expr)` where `expr` is a statically evaluable expression of type `integer {4, 8}`) may be used as part of a return type, so return types may be bitvectors which are not fixed width.

See rule LJBG for why `bits(-: integer{4, 8})` and `bits(integer{4, 8})` have an undetermined width.

See rule NCNQ for why `bits(N)`, where `N` is an under-constrained integer, is an under-constrained width bitvector that has a determined width.

See rule SKRK for why `bits(expr)` where `expr` is a statically evaluable expression of type `integer {4, 8}` is a constrained width bitvector of determined width.

Examples of valid return types:

```
func declare{N}(b: bits(N)) => bits(2 * N)
begin
    var b1: bits(2 * N);

    return b1;
end

let exp: integer{4, 8} = 4;

func decl() => bits(exp)
begin
    var b: bits(exp);

    return b;
end
```

I$_{\text{GLWM}}$ When used as part of the type of a storage element's declaration, a constrained width bitvector of undetermined width allows that part of the storage element's width to be implicitly determined by the initialization expression.

I$_{\text{MTWL}}$ When a constrained width bitvector is used as part of a formal argument, the combination of the domain rule for constrained width bitvectors and the type satisfaction rule are used to determine whether invocations have legal actual arguments.

7.5.7 Use of bitvector storage elements and expressions

I$_{\text{EOAX}}$ A checked type conversion on the width expression of a bitvector type is sufficient to ensure the width is a constrained integer.

I$_{\text{NHXT}}$ In many common cases, integer expressions will cause suitable constrained type to be implicitly used. See also 7.8.5 *Primitive operations on integers*. For example, the width `configWid+gExtra` has an acceptable constraint in the following code:

```
// The initializer of config storage elements may be overridden before
// execution-time initializer expressions are evaluated.
config configWid: integer {32, 64}  = 32;
config gExtra   : integer {0, 8}    = 0;
```

```
// Since configWid is a constrained integer we can declare:
var gReg: bits(configWid); // i.e. bits({32,64})

var gRegF: bits(configWid+gExtra); // i.e. bits({32,40,64,72})
// The expression (configWid+gExtra) is an integer {32,40,64,72}
```

I_{BTCY} Note that compile-time-constant integers are implicitly constrained, hence the addition of a constraint to a bitvector width is unnecessary for compile-time-constant widths. See also *7.7.5 Implicit constraints for compile-time-constant integer expressions*.

7.5.8 Examples of constrained width bitvectors

I_{CGYH} The following shows some declarations of bitvectors:

```
var gWid: integer {8,16,32};

// Note that a and b are "formal arguments" but are not "parameters"!
func prod(a: integer, b: integer) => integer
begin
    return (a*b);
end

func declarations()
begin
    let N = gWid;
    // N is an execution-time immutable constrained integer
    // The implicit type of N is integer {8,16,32}

    var A: bits(N);
    // A is a constrained width bitvector of determined width N
    // whose width is constrained to be IN {8,16,32}

    var B: bits(N * 2);
    // B is a constrained width bitvector of determined width N*2
    // whose width is constrained to be IN {16,32,64}

    // The following contrived example shows that even if
    // prod is a trivial function, the result is unconstrained
    let prodWid = prod(N,2); // implicitly an integer
    var C: bits(prodWid);
    // Illegal since prod(N,2) is an unconstrained integer

    var D: bits(prodWid as {16,32,64});
    // Legal but requires an execution-time check that (prod(N,2) IN {16,32,64})
    // This check may be elided by the compiler
    // D is a constrained width bitvector of determined width prodWid
    // whose width is constrained to be IN {16,32,64}

    var E: bits(N as {64,128}); // Illegal - fails type-check
    // Requires (N IN {64,128})
    // This can be determined as false at compile time since
    // the type of N is integer {8,16,32}.
end
```

Note

We allow:

```
func bitProd(a: integer, b: integer)=> bits(a*b)
```

so it has been suggested that we could allow:

```
func prod(a: integer, b: integer)=> integer {a*b}
```

such that if the actuals for a and b are well-constrained integers in the invocation then the constraint on the result type follows the rules for constraint calculations with primitive operators.

This may be helpful for type checking at the invocation location. For example, it would make the declaration of `var C` above legal.

I_{QNSD} The following shows the use of some constrained width bitvectors:

```
type widTy of integer {4,8};

func callCB {N: widTy} (arg1: bits(N)) => bits(N)
begin
    // The formal argument is a constrained width bitvector
    // whose determined width N is IN {4,8}
    // For checking actual arguments, the domain of arg1 is
    // domain(bits(4)) union domain(bits(8))
    return arg1;
end

func bitvector(N: widTy)
begin
    var CB1: (bits(-: widTy), integer) = (Zeros(N), 0);
    // CB1 is a tuple whose first element is a constrained width bitvector
    // whose width is implicitly determined by the width of the bitvector
    // returned by the call to Zeros, i.e. N.
end
```

I_{FPVZ} Note that the rules of 7.5.6 *Use of bitvectors of undetermined width* mean that there is a potentially confusing asymmetry in the use of `bits(-: ty)` as illustrated by the following example:

```

var none: bits(-: integer {1});
// legal since this is a fixed width bitvector whose determined width is 1
// not a constrained width bitvector of undetermined width

var many: bits(-: integer {1,2});
// illegal since this is a constrained width bitvector of undetermined width
// and there is no initialization expression

config configValue: integer {1,2} = 1;
var many: bits(-: integer {1,2}) = Zeros(configValue);
// legal since this is a constrained width bitvector of undetermined width
// and this is a declaration where an initialization expression is given
```

7.6 Relations on types

In this section we describe the main relations used in determining whether a type is suitable at its point of use.

7.6.1 Subtype-satisfaction

We define the *subtype-satisfaction* relation as follows:

D_{TRVR} A type `T` subtype-satisfies type `S` if and only if all of the following conditions hold:

- If `S` has the structure of an integer type then `T` must have the structure of an integer type.
- If `S` has the structure of a real type then `T` must have the structure of a real type.
- If `S` has the structure of a string type then `T` must have the structure of a string type.
- If `S` has the structure of a boolean type then `T` must have the structure of a boolean type.
- If `S` has the structure of an enumeration type then `T` must have the structure of an enumeration type with exactly the same enumeration literals.
- If `S` has the structure of a bitvector type with determined width then either `T` must have the structure of a bitvector type of the same determined width or `T` must have the structure of a bitvector type with undetermined width.
- If `S` has the structure of a bitvector type with undetermined width then `T` must have the structure of a bitvector type.
- If `S` has the structure of a bitvector type which has bitfields then `T` must have the structure of a bitvector type of the same width and for every bitfield in `S` there must be a bitfield in `T` of the same name, width and offset, whose type type-satisfies the bitfield in `S`.
- If `S` has the structure of an array type with elements of type `E` then `T` must have the structure of an array type with elements of type `E`, and `T` must have the same element indices as `S`.
- If `S` has the structure of a tuple type then `T` must have the structure of a tuple type with same number of elements as `S`, and each element in `T` must type-satisfy the corresponding element in `S`.
- If `S` has the structure of a record type then `T` must have the structure of a record type with at least the same fields (each with the same type) as `S`.
- If `S` has the structure of an exception type then `T` must have the structure of an exception type with at least the same fields (each with the same type) as `S`.
- If `S` does not have the structure of an aggregate type or bitvector type then the domain of `T` must be a subset of the domain of `S`.
- If either `S` or `T` have the structure of a bitvector type with undetermined width then the domain of `T` must be a subset of the domain of `S`.

I_{SJDC} Note that the condition on bitvector widths applies to bitvectors with or without bitfields.

I_{MHYB} Note that subtype-satisfaction alone does not make named type `T` a subtype of named type `S`. `T` must also be declared to be a subtype of `S` or one of the subtypes of `S` - see also 4.3.1 *Subtypes*.

7.6.1.1 Subtype-satisfaction and bitvectors

I_{TWTZ} If both `S` and `T` are bitvectors with determined width then we require their width to match. For example you cannot use `bits(2)` where a `bits(4)` is required.

I_{GYSK} Where `S` is a bitvector with determined width and `T` has undetermined width, the domain requirement will stop `T` from being used where `S` is expected. For example, `bits({2,4})` does not subtype-satisfy `bits(4)` since the domain of `bits({2,4})` is not a subset of the domain of `bits(4)`.

However, since the checked type conversion `bits({2,4})as bits(4)` only fails the subtype-satisfaction because of the domain rule, but the domains do intersect, a runtime check will be inserted. See also 7.13.1 *Checked type conversions on expressions*.

I_{KXSD} The rules of subtype-satisfaction permit a bitvector of undetermined width to subtype-satisfy a bitvector of determined width if its domain is a subset of the determined width bitvector's. We will refer to this as *undetermined width satisfaction*.

Note that the only legal undetermined width bitvector is `bits(constraint)` (see *7.5.4 Summary of types of bitvector*). This can only arise in three ways (see *7.5.6 Use of bitvectors of undetermined width*)

1) As part of the type of a formal argument of a subprogram declaration.

Since formal arguments are immutable we will never have an assignment to them, so undetermined width satisfaction is not applicable. Note that `bits(constraint)` allows concise declarations of subprograms by removing the need for a parameter declaration where the width is not required in the subprogram body.

2) As the type of a storage element with an initialization expression.

The initialization expression must have the type of a determined width bitvector. The type checker will know the storage element's width from the initialization expression, hence undetermined width satisfaction will not apply to assignments to it. The notation `bits(constraint)` simply allows concise declarations.

3) As part of a Checked Type Conversion

Undetermined width satisfaction allowing `bits(constraint)` to subtype-satisfy `bits(wid)` is intended to allow the Checked Type Conversion rules to work, see bullet 2 of Rule PZZJ.

Checked Type Conversions of this kind are required to allow assignment of a undetermined width bitvector formal arguments to storage elements (which have determined width). For example:

```
func ctcOfFormal(arg: bits({2,4}), cond: boolean)
begin
    var dst: bits(4);
    if (cond) then
        dst = arg as bits(4);
        // bits({2,4}) does not subtype-satisfy bits(4)
        // but only because of the domains
        // so an execution time check is inserted
    end
end
```

I_{KNXJ} For the case where `S` is a bitvector type with undetermined width, `S` will have some constraints which define its domain. Any bitvector whose domain is a subset of the domain of `S` will type satisfy `S`.

For example `bits(2)` subtype-satisfies `bits({2,4,8})` due to the domain rule, but `bits(1)` does not subtype-satisfy `bits({2,4,8})`. Similarly, `bits({2,4})` subtype-satisfies `bits({2,4,8})` due to the domain rule, but `bits({1,2})` does not subtype-satisfy `bits({2,4,8})`.

However, since the checked type conversion `bits({1,2})as bits({2,4,8})` only fails the subtype-satisfaction because of the domain rule, but the domains do intersect, a runtime check will be inserted. See also *7.13.1 Checked type conversions on expressions*.

I_{HSWR} Where both `S` and `T` are bitvectors of determined width, since their widths must match, we do not need to compare their domains.

7.6.2 Type-Satisfaction

We define the *type-satisfaction* relation as follows:

R_{FMXK} Type `T` type-satisfies type `S` if and only if at least one of the following conditions holds:

- `T` is a subtype of `S`
- `T` subtype-satisfies `S` and at least one of `S` or `T` is an anonymous type
- `T` is an anonymous bitvector with no bitfields and `S` has the structure of a bitvector (with or without bitfields) of the same width as `T`.

I_{NLFD} Note that every type `T` is a subtype of itself - see rule NXRX.

7.6.3 Type-clashing

We define the *type-clashing* equivalence relation as follows:

D$_{\text{VPZZ}}$ A type `T` type-clashes with `S` if any of the following hold:

- they both have the structure of integers
- they both have the structure of reals
- they both have the structure of strings
- they both have the structure of enumeration types with the same enumeration literals
- they both have the structure of bit vectors
- they both have the structure of arrays whose element types type-clash
- they both have the structure of tuples of the same length whose corresponding element types type-clash
- `S` is either a subtype or a supertype of `T`

I$_{\text{PQCT}}$ Note that `T` subtype-satisfies `S` implies that `T` and `S` type-clash.

I$_{\text{WZKM}}$ Note that type-clashing is an equivalence relation. In particular note that if `T` type-clashes with `A` and `B` then `A` and `B` type-clash.

7.6.4 Subprogram clashing

D$_{\text{BTBR}}$ Two subprograms *clash* if all of the following hold:

- they have the same name
- they are the same kind of subprogram
- they have the same number of formal arguments
- every formal argument in one type-clashes with the corresponding formal argument in the other

I$_{\text{FSFQ}}$ Whether or not the return types of the subprograms type clash is ignored when determining clashing.

I$_{\text{SCTB}}$ Note that the formal arguments of a setter include the RHS argument. See 4.5.3 *Getters and Setters*.

I$_{\text{PFGQ}}$ In most cases, if two subprograms have the same name they must be of the same kind due to the rules of 4.1 *Scope of global declarations*, except where one is a getter and the other is a setter.

Hence, although a procedure named `f` does not clash with a function named `f` since they are different kinds, their declaration is illegal due to global scope rules.

A getter named `s` does not clash with a setter named `s`, even if they have similar formal arguments, since they are not of the same kind. Hence their declaration is legal.

R$_{\text{PGFC}}$ It is illegal to declare a subprogram if it clashes with any other declared subprogram.

7.7 Type checking rules

We now provide the rules for checking that various uses of typed expressions are suitable at their point of use.

D$_{\text{JRXM}}$ A *type environment* is a mapping from identifiers to their types.

I$_{\text{ZTMQ}}$ This includes identifiers which denote types, subprograms, storage elements etc.

7.7.1 Global type checking

D$_{\text{HBCP}}$ The *global type environment* is a type environment which maps every globally declared identifier to its type.

I$_{\text{SMMH}}$ Since named type identifiers are globally declared, and since they may depend non-recursively on other named types, it should be possible to add named types to the global type environment using a simple recursive algorithm.

I$_{\text{DFML}}$ Note that determining the type of a global identifier may require the type of its initialization expression to be determined first. That type may not depend recursively on itself.

R$_{\text{FWQM}}$ When determining types of global declarations which are not subprogram declarations, only the identifiers mapped by the *global type environment* shall be used.

7.7.2 Subprogram type checking

R$_{\text{HWTV}}$ When type-checking a subprogram declaration begins, the current type environment is initialized with the contents of the global type environment and also all identifiers introduced into the subprogram's local scope by rules PTDD and JBXS.

R$_{\text{SCHV}}$ When determining types within a subprogram declaration, only the identifiers mapped by the *current type environment* shall be used.

R$_{\text{VDPC}}$ A subprogram declaration is type checked by type checking each statement in its declaration in the order they appear in the declaration.

R$_{\text{YSPM}}$ Local declarations of `variable`, `let` or `constant` identifiers introduce a new identifier into the current type environment which maps to the type of the identifier.

R$_{\text{JBXQ}}$ where a new `stmt_list` is encountered, a new type environment is created which is initialized with the contents of the current type environment. This new type environment becomes the current type environment. When the end of the `stmt_list` is encountered, the current type environment is discarded and the type environment which was current at the start of the `stmt_list` becomes the current type environment again.

7.7.3 Statement type checking

R$_{\text{FTPK}}$ A value of type `T` may be used in a return statement if and only if `T` type-satisfies the enclosing function's return type.

R$_{\text{JQYF}}$ The expression in an `assert` statement must have the structure of `boolean`.

R$_{\text{NXRC}}$ The expression in a `throw` statement must have the structure of `exception`.

R$_{\text{NBDJ}}$ All condition expressions in conditional statements must have the structure of `boolean`.

R$_{\text{WGSY}}$ The case guard of a case statement's `when` clause must have the structure of `boolean`.

R$_{\text{VTJW}}$ The start and end expressions in a `for` statement must have the structure of `integer`.

R$_{\text{FTVN}}$ The condition expression in a `while` or `repeat` statement must have the structure of `boolean`.

R$_{\text{SDJK}}$ The type given in a `when` clause of a `try`..`catch` statement must have the structure of `exception`.

R$_{\text{WVXS}}$ The identifier given in a `when` clause of a `try`..`catch` statement is introduced into the new type environment which is created by rule JBXQ when the body of the `when` clause is encountered. The identifier maps to the type it is annotated with in the `when` clause.

I$_{\text{FCGK}}$ Note that the identifier given in the `when` clause does not map to the type of the matched exception.

7.7.4 Assignment and initialization type checking

The following rules apply to all storage elements, whether global or local.

R$_{\text{WMFV}}$ The type of a storage element is the same as the type of the identifier which denotes the storage element.

R$_{\text{GNTS}}$ It is illegal for a storage element whose type has the structure of the under-constrained integer to be assigned a value whose type has the structure of the under-constrained integer.

R$_{\text{ZCVD}}$ It is illegal for a storage element whose type has the structure of the under-constrained integer to be initialized with a value whose type has the structure of the under-constrained integer, unless the type is omitted from the declaration (and therefore the type can be unambiguously inferred) or the initialization expression is omitted (and therefore the type is not omitted from the declaration).

R$_{\text{LXQZ}}$ A storage element of type `S`, where `S` is any type that does not have the structure of the under-constrained integer type, may only be assigned or initialized with a value of type `T` if `T` type-satisfies `S`

I$_{\text{MMKF}}$ The puprpose of rules ZCVD and GNTS is to disallow examples like the following:

```
func foo{N, M}(bv1 : bits(N), bv2 : bits(M))
begin
    var a : integer{0..N};
    var b : integer{0..M};
    a = b; // illegal
end
```

In the example above, it is not evident whether the type `integer{0..N}` should type-satisfy `integer{0..M}` or vice versa.

Also consider the following example:

```
func foo{N}(bv : bits(N))
begin
    let a = N;           // legal, type is inferred
    var b : integer{N}; // legal, type is explicit and initialization value is omitted
    b = a;               // illegal
end
```

In the declaration statements of the example above, the types of the storage element and the initializing values are identical by definition. The assignment however is still illegal due to GNTS.

I$_{\text{YYQX}}$ Due to rules YDFQ and ZFFV the use of a `setter` or `getter` is type checked according to the rules for subprogram invocation.

I$_{\text{DGWJ}}$ Note that we maintain strong typing of named types since FMXK only allows named types to be assigned to each other if `T` subtypes `S` which requires explicit declaration. See also information statement MHYB.

I$_{\text{KKCC}}$ Note that type-satisfaction allows assignment between a type and many anonymous types including constrained types.

7.7.5 Implicit constraints for compile-time-constant integer expressions

R$_{\text{ZJKY}}$ The type of an integer compile-time-constant expression is the constrained integer type whose constraint holds only the value of the expression.

I$_{\text{RYRP}}$ Implicit constraints allow assignment of compile-time-constant integer expressions to constrained integers at compile-time without an explicit checked type conversion.

EXAMPLE

```
var intWid: integer {32,64} = 32;
// legal since the type of 32 is integer {32}
```

7.8 Subprograms and overloading

7.8.1 Dependently typed bit vector formals

I$_{PDKT}$ ASL permits the declaration of subprograms with formals which are parameterized such that where a formal or return type is a bitvector, its width may depend on the value of the subprogram parameters.

I$_{BLVP}$ As required by rule JBXS, a parameter of a subprogram shall not be declared both in the parameter list and in the formal list.

R$_{LVTH}$ Checked type conversions shall not be used on bitwidth expressions in subprogram formals or subprogram return types.

I$_{TBHH}$ It is illegal to provide a constraint on a parameter anywhere other than as part of its declaration. Any practical case where a constraint is required for a formal or a return type are covered by specifying a constrained integer parameter.

I$_{ZLZC}$ All the constraints in the formals in following subprogram declaration are illegal:

```
func illegal_constraints {N: integer {8,16,32}} (
    arg0: bits(N as {8,16,32}),
    arg1: bits((N+1) as {9,17})
    )
begin
    return;
end
```

R$_{TJKQ}$ Any parameter of a subprogram which is declared as the unconstrained integer type shall be treated as though it was declared as the under-constrained integer.

I$_{LFJZ}$ This is safe to do since every invocation must provide a constrained width bitvector (see also 7.8.6 *Primitive operations on bitvectors*.)

I$_{RKBV}$ Unconstrained integer parameters are treated as under-constrained integer parameters both within the subprogram and at invocations.

I$_{TQGH}$ Every parameter is a constrained integer as far as type-checking is concerned.

R$_{RHTN}$ The width of any part of a formal argument or return type which is a bitvector type must be a constrained integer.

I$_{BZVB}$ The rules in 7.8.5 *Primitive operations on integers* mean that for most expressions, the width of a bitvector in a formal argument or return type will be at least the under-constrained integer. For example, a width which is given the sum of two parameters is an under-constrained integer.

I$_{RQQB}$ A formal argument is not a parameter unless it is declared as a parameter of the subprogram or is declared as a formal and is also used in the type of a bitvector formal argument or the return type.

```
func testParams{K}(N: integer, M: integer, L: integer, lbv: bits(L), kbv: bits(K) ) => bits(N
    ↪)
begin
    var kBits: bits(K); // legal
    // K is a parameter, so it is an under-constrained integer

    var nBits: bits(N); // legal
    // N is a parameter because it is a formal argument and is used in the type of the
        ↪bitvector of a return type, so it is an under-constrained integer

    var lBits: bits(L); // legal
    // L is a parameter because it is a formal argument and is used in the type of the
        ↪bitvector of a formal argument, so it is an under-constrained integer

    var mBits: bits(M); // ILLEGAL
    // M is not a parameter so it is an unconstrained integer

    return nBits;
```

```
end
```

7.8.1.1 Examples

Consider this declaration of a function called `ones`:

```
func ones(N: integer) => bits(N)
begin
    // N is a parameter so it is declared as an under-constrained integer.
    // The return type is therefore the
    // under-constrained width bitvector of undetermined width.
    // If N were left as `integer` instead of being declared as an
    // under-constrained integer then the following would fail since it is
    // illegal to declare `bits(expr)` if expr is an unconstrained integer.
    var ans: bits(N) = Zeros(N);
    // for any invocation, the returned bitvector's width has the same
    // constraint as the actual argument which provides the parameter value.
    // For example:
    // If N is 1 in the invocation,
    // then the returned value is bits(1: integer{1}).
    // If N is an `integer {4,8}` in the invocation
    // then the returned value is bits({4,8})
    return (NOT ans);
end
```

The rules for domains of constrained integers in 7.4 *Constrained Integers* mean that an invocation with an unconstrained integer actual argument will not satisfy the formal argument `N` which is actually declared as the under-constrained integer.

For example, consider the following:

```
config myWid: integer = 5;
// myWid is an unconstrained integer

func test()
begin
    var arg = ones(myWid);

    // Fails type satisfaction test since the domain of an
    // unconstrained integer is not a subset of the
    // under-constrained integer.
end
```

The following examples relate to the use of parameters in return types.

Note that in general, a bitvector return type must have a width which can be determined at the invocation.

Example 1:

Parameters must have values:

```
func getDefault {N}() => bits(N) // illegal
begin
    // There is no parameter-defining type in the formals
    // so return width cannot be determined at invocation
    return ones(Zeros(N));
end
```

Example 2:

Parameters must have values, even if they are not used in the subprogram:

```
config configWid: integer {8,16} = 8;

func getDefault {N}() => bits(N)
begin
    // There is no parameter-defining type in the formals
    // so return width cannot be determined at invocation
    return ones(configWid);
end
```

Example 3:

A more specific example, where the return width is not a parameter.

```
config configWid: integer {8,16} = 8;

func getDefault () => bits(configWid)
begin
    // return type is well-constrained width bitvector
    // of determined width configWid
    return ones(configWid); // Legal
end
```

In summary:

- a parameter `N` must occur as a formal argument or in a width expression in a formal argument, otherwise it would not be a "parameter"
- If `N` is in a formal argument's bit width then the invocation will involve a bitvector whose width must be a constrained integer, so `N` will be constrained at point of invocation.
- If the only type in the subprogram signature involving `N` is the return type, then `N` must be an argument as well as a parameter. As noted above, no invocation can provide an unconstrained integer actual for `N`, since `N` is treated as the under-constrained integer.

7.8.2 Subprogram invocations

D_{CFYP} A function invocation consists of an `identifier` which denotes the name of the invoked function, followed by a parenthesised list of zero or more expressions which denote the *actual arguments* of the invocation. See syntax in 5.4 *Atomic expressions*.

D_{VXKM} A procedure call consists of an `identifier` which denotes the name of the called procedure, followed by a parenthesised list of zero or more expressions which denote the *actual arguments* of the call. See syntax in 6.1 *Statements*

D_{LJLW} A getter invocation consists of an `identifier` which denotes the name of the invoked getter, either on its own, or followed by a bracketed list of zero or more expressions which denote the *actual arguments* of the call. See syntax in 5.4 *Atomic expressions* and related rules in 5.9 *Arrays, bitslices and invoking getter functions*.

D_{MFBC} A setter invocation consists of an assignment where:

- part of the LHS consists of an `identifier` which denotes the name of the invoked setter, either on its own, or followed by a bracketed list of zero or more expressions which denote the *actual arguments* of the call.
- the corresponding part of the RHS consists of an expression which denotes the actual argument corresponding to the setter's RHS argument (see 4.5.3 *Getters and Setters*)

R_{WHRS} If the declared type of a setter's RHS argument has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

R_{TCDL} Where a parameter is also a formal argument, the parameter's value and constraints for an invocation of the subprogram are the same as the corresponding actual in the invocation.

I_{VFDP} When type checking actual arguments in an invocation which correspond to a formal argument which is also a parameter, the constraints in the subprogram *declaration* for the formal argument are used, as is the case for any formal argument.

D_{TRFW} A bitvector type in a formal argument may be *parameter-defining* for a parameter.

R_{KMBD} A bitvector type in a formal argument is parameter-defining for a parameter if its bitwidth is the value of that parameter.

I_{YMHX}

I$_{\text{YMHX}}$ For example, `bits(M)` is parameter-defining for `M` but `bits(M+1)` is not.

R$_{\text{CCVD}}$ Where a parameter is not a formal argument, the subprogram must have at least one formal argument which is parameter-defining for that parameter.

Example

```
var global: integer {4,8};

func foo {parm} (
    arg0: bits(global),
    arg1: bits(parm+global),
    arg2: (integer, bits(parm))
    )
begin
    // The type of the second part of the tuple `arg2` is
    // parameter-defining for `parm`, without which the
    // declaration would be illegal.
    // None of the other formals are parameter-defining.

    return;
end
```

R$_{\text{QYBH}}$ Where a parameter is not a formal argument, its value and constraints are taken from the actuals which correspond to the parameter's parameter-defining formals.

R$_{\text{PFWQ}}$ An invocation is illegal if it provides different values or constraints for a particular parameter's parameter-defining formals.

R$_{\text{ZLWD}}$ Where a parameter is not a formal argument, the declared type of the parameter must be type-satisfied by an integer type with the constraints taken from the invocation's actuals.

I$_{\text{FLKF}}$ When a parameter takes its value from one of the actual argument's bitwidths, type satisfaction of all other formals by the actuals will ensure that occurrences of the parameter in all other formal bitwidth expressions comply with that value.

D$_{\text{PMBL}}$ The formal arguments and return type of a subprogram have an *invocation type* for each invocation.

R$_{\text{MWBN}}$ The invocation type of a formal argument or return type is its declared type after the values and constraints of parameters have been applied.

R$_{\text{TZSP}}$ A subprogram invocation matches a subprogram declaration if all of the following hold:

- the name of the invoked subprogram matches the name of the declared subprogram
- every formal argument's declared type is type-satisfied by its invocation type (Ed: now redundant?)
- every formal argument's invocation type is type-satisfied by the corresponding actual argument
- if the subprogram has a return type then it is type satisfied by its invocation type

I$_{\text{SBWR}}$ When determining whether a formal bitvector argument's declared type is type-satisfied by its invocation type, parameter values are known, so the declared type, the invocation type and the actual's type will all be bitvectors of determined width, hence subtype-satisfaction will not compare their domains.

I$_{\text{CMLP}}$ Note that integer formal arguments which are also parameters of the subprogram are treated as the under-constrained integer and therefore cannot be type-satisfied by an unconstrained integer actual.

R$_{\text{BQJG}}$ The type of a function of getter invocation is the invocation type of its return type.

I$_{\text{BTMT}}$ In general, clashing subprograms cannot be usefully declared since one of the subprogram's formals type-satisfies the other subprogram's formal, hence an actual of that type will type-satisfy both subprograms.

R$_{\text{RTCF}}$ It is a type-checking error if a subprogram invocation does not match exactly one subprogram declaration.

See also 4.5.2 *Functions and procedures*.

7.8.2.1 Example of overloading

```
type shape of integer;
type square of integer subtypes shape;

var myShape: shape;
var mySquare: square;

// following declarations of f are illegal since
// both have the structure of an integer so their argument type-clashes
func f(x: shape)
begin
    pass;
end

func f(y: square)
begin
    pass;
end

func f_test()
begin
    // if the declarations were legal, the which f would be invoked?
    f(mySquare);
end

func g(x: shape,  y: integer)
begin
    pass;
end

func g(x: square, y: real)
begin
    pass;
end

func g_test()
begin
    g(mySquare, 0);  // legal
    // because since the first declaration of g has
    // a first formal of type shape with is type-satisfied by the subtype square
    // and a second formal which is type satisfied by an integer.

    g(myShape, 0.1); // illegal
    // because no declaration of g has
    // a first formal which is type-satisfied by a shape
    // and a second formal which is type-satisfied by a real.
end

// following declarations of h are illegal
// since all arguments type-clash with corresponding arguments in the other declaration
func h(x: shape, y: square)
begin
    pass;
end

func h(x: square, y: shape) // Illegal
begin
    pass;
end

func h_test()
begin
    // if the declarations were legal, the which h would be invoked?
    h(mySquare, mySquare); // either h may apply!!
end
```

7.8.2.2 Example: Actual parameter type-satisfaction

```
func f (N: integer {2,4}, arg0: bits(N), arg1: bits(N))
begin
    // the parameter N is also a formal argument so it will take its value
    // and constraints from the corresponding actual parameter
    return;
end

func test {wid: integer {2,4,8}} (bus: bits(wid))
begin
    f (wid, bus, bus); // illegal
    // wid does not type satisfy the invocation type of N which is integer {2,4}
end
```

7.8.2.3 Example: Parameter from actual width type-satisfaction

```
func f {N: integer {2,4}} (arg0: bits(N), arg1: bits(N*2))
begin
    // N is not a formal argument so it takes its value from the actual
    // corresponding to arg0 since the type of arg0 is a
    // parameter-defining bitvector
    return;
end

func test {wid: integer {2,4,8}} (bus: bits(wid))
begin
    f (bus, [bus,bus]); // illegal
    // N takes the value `wid` which is an integer {2,4,8}
    // the declared type of the parameter 'N' is not type-satisfied by that type

    let littleBus = bus as bits({2,4}); // type checker knows width of littleBus == wid
    f(littleBus, [littleBus, bus]); // legal
    // N takes the value equal to the width of littleBus which is an integer {2,4}
    // the declared type of the parameter 'N' is type-satisfied by that type
    // The declared type of arg0 is bits(N as {2,4}) which is type satisfied by
    // its invocation type bits(width of littleBus as {2,4})
    // since the type checker knows that N == width of littleBus as {2,4}
    // The two actuals type satisfy the formal's invocation types which are
    // arg0: bits(width of littleBus)
    // arg1: bits(width of littleBus *2)
end
```

7.8.2.4 Example: Parameter-defining types

Parameter N is not a formal argument in the following functions, so it must appear alone as a bitwidth in at least one formal argument.

```

func f1 {N} (arg: bits(N)) => bits(N+1) // legal
begin
    return [arg, '0'];
end

func f2 {N} (arg: bits(N+1)) => bits(N) // illegal
begin
    // No parameter-defining type for N in the formals
    // This could be handled with more complex rules, but is currently illegal
    return arg[N-1:0];
end

func f3 {N, M} (arg: bits(N+M)) => bits(M) // illegal
begin
    // Clearly impossible to handle
    // For 'f3(Zeros(8))' what is N? How wide is the result??
    return arg[M-1:0];
end

func f4 {N, M} (arg1: bits(M+N), arg2: bits(N)) => bits(N) // illegal
begin
```

```
    // No parameter-defining type for M in the formals
    // This could be handled with more complex rules, but is currently illegal
    return (arg1[N-1:0] + arg2);
end

func f5 (N: integer, M: integer, arg1: bits(N+M), arg2: bits(N)) =>  bits(N) // legal
begin
    // parameters are formal arguments
    return (arg1[N-1:0] + arg2);
end
```

7.8.2.5 Example: Invocation constraints on parameters

IKTJN The result of a subprogram invocation may be more constrained than the declaration indicates. For example:

```
func sameWid {N: integer {2,4,8,16}} (A: bits(N), B: bits(N)) => bits(N)
begin
    return A;
end

func test1()
begin
    var A1, B1: bits(8);
    var C1 = sameWid(A1, B1);
    // The invocation type of sameWid's return type is bits(8)
    // so `C1` has type `bits(8)`
end

func test2(N: integer{4,8})
begin
    let wid: integer {2,4,8} = N; // A little unusual...
    var A2: bits(N);   // bits(N as {4,8})
    var D2: bits(wid); // bits(wid as {2,4,8})
    A2 = D2; // legal - matching determined widths
    D2 = A2; // legal - matching determined widths
    // Although A2 and D2 have the same width, they have different
    // constraints so the following is illegal
    var result = sameWid(A2, D2);
    // If it were not illegal then the return type could not be determined
    // and is either bits(N as {4,8}) or bits(wid as {2,4,8})
end
```

7.8.2.6 Example: Invocation constraints on parameters:

```
func f(P: integer {2,4,8}, M: integer {4,8,16})
begin
    // P==>(P as {2,4,8})
    // M==>(M as {4,8,16})

    // Note the invocations of Zeros here:
    var opA: bits(P) = [Zeros(P-1), '1']; // opA is bits(P from {2,4,8})
    var opB: bits(M) = [Zeros(M-1), '1']; // opB is bits(M from {4,8,16})

    if (P==M) then
        // var dst = opA+opB; // ILLEGAL
        // operation `+` is (bits(N), bits(N)) => bits(N)
        // but opA and opB are bitvectors of
        // potentially different widths P and M
        // so they do not agree on the value of N

        var dst = opA+(opB as bits(P));
        // The operands now agree that N is `P`
        // an execution-time check that opB is `bits(N)` is required
        // The guard above allows a tool to elide this check.
    end
end
```

Note that `(P-1)` is of type `integer {1,3,7}` so the invocation of `Zeros(P-1)` returns a constrained width bitvector `bits({1,3,7})` of width `P-1`. This is concatenated with a bitvector of width `1` resulting in a bitvector of width `(P-1)+1` which is an `integer {2,4,8}`.

That means the resulting bitvector is a constrained width bitvector `bits({2,4,8})`.

The assignment is also acceptable since the widths of the LHS and RHS are equal. Hence no checked type conversion is required, just as we would not expect one for the following:

```
var opC: bits(M) = Zeros(M);
```

7.8.2.7 Example: Function with under-constrained integer formal

```
func underconstrained_formal(wid: integer) => bits(wid)
begin

    // wid is a parameter so it is treated as an underconstrained
    // integer. The return type is therefore the underconstrained width
    // bitvector of determined width `wid`.

    // If wid were left as `integer` instead of being treated as an
    // underconstrained integer then the following would fail since it is
    // illegal to declare `bits(expr)` if expr is an unconstrained
    // integer.

    var ans: bits(wid);

    // The following works because the argument of Zeros is also an
    // under-constrained integer so the formal and actual have the
    // same domains

    ans = Zeros(wid);

    // For any invocation of underconstrained_formal, the returned
    // bitvector's width has the same constraint as the actual argument
    // due to the rules about invocation widths

    // For example:
    // If wid is 1 in the invocation,
    // then the returned value is bits(1: integer{1}).
    // If wid is an `integer {4,8}` in the invocation
    // then the returned value is bits({4,8})
    return (NOT ans);
end


// Three cases for invocation of function `underconstrained_formal`
// which has an under-constrained integer formal

func legal_fun_constrained_actual (N: integer {1,2}) => bits(N)
begin
    // This invocation is OK because {1,2} is a subset of the domain of the underconstrained
        ↪integer
    // The result of the invocation is a constrained width bitvector,
    // of determined width `N` with constraint {1,2}
    // since the invocation width of the return type of `underconstrained_formal` is `N`
    return underconstrained_formal(N);
end

func legal_fun_underconstrained_actual (N: integer) => bits(N)
begin
    // The following invocation is OK because the domain of the
    // underconstrained integer `N` in the invoking function is the
    // same as the domain of the under-constrained integer parameter
    // `wid` in the invoked function.
    // The result of this invocation of `underconstrained_formal` is an
    // under-constrained width bitvector of determined width `N`
    // since the invocation width of the return type of `underconstrained_formal` is `N`
    return underconstrained_formal(N);
end

func illegal_fun_unconstrained_actual (arg: integer)
begin
    // Illegal invocation of `underconstrained_formal` since the domain of the unconstrained
    // integer (arg) is not a subset of the domain of the under-constrained integer (wid)
    let - = underconstrained_formal(arg);

    // Note that if the formal was *only* treated as an under-constrained
    // integer *within* the subprogram then it would be an integer at the
    // invocation and this would be legal since the domain of the actual
    // integer is equal to the domain of the formal integer
end
```

7.8.2.8 Example: Function with well-constrained integer formal

```
func constrained_formal (N: integer {1,2,3}) => bits(N)
begin
    return Zeros(N);
end

// Cases for invocation of function
// which has a well-constrained integer formal

func legal_fun_constrained_actual (N: integer {1,2}) => bits(N)
begin
    // This invocation is OK because {1,2} is a subset of {1,2,3}
    // The result of the invocation is a constrained width bitvector,
    // of determined width `N` with constraint {1,2}
    // since the invocation width of the return type of constrained_formal is `N`
    return constrained_formal(N);
end

func illegal_fun_constrained_actual (N: integer {2,3,4}) => bits(N)
begin
    // Illegal because {2,3,4} is NOT a subset of {1,2,3}
    // See Rule TZSP clause 2
    // The invocation width of the return type of constrained_formal is `N`
    return constrained_formal(N); // requires a checked type conversion
end

func illegal_fun_underconstrained_actual (N: integer) => bits(N)
begin
    // Illegal since the domain of the under-constrained integer is
    // NOT a subset of {1,2,3}
    return constrained_formal(N); // requires a checked type conversion
end

func illegal_fun_unconstrained_actual (arg: integer)
begin
    // Illegal since the domain of the unconstrained integer is NOT a
    // subset of {1,2,3}
    let illegal = constrained_formal(arg); // requires a checked type conversion
end
```

7.8.2.9 Example: Assignments with under-constrained integers

```
// Cannot assign to an under-constrained integer
// since it is always an immutable parameter!
// Two cases for assignments from an under-constrained integer

func assign(N: integer) => bits(N) // N is an under-constrained integer
begin
```

```
    let legal1 = N; // legal1 is an under-constrained integer equal to N

    // legal since the domain of the under-constrained integer is a
    // subset of Domain(integer)
    let legal   : integer = N;

    // illegal since the domain of the under-constrained integer is
    // NOT a subset of Domain(integer{32, 64})
    let illegal : integer {32, 64} = N;

    return Zeros(N);
end
```

7.8.2.10 Example: Precise checked type conversions on actuals

In the following examples, it is not enough to provide a constrained width bitvector for the checked type conversions on actuals. Instead, we must provide a checked type conversion which ensures the actuals are the same type, i.e. they have the same width.

```
func unforcedBits {N} (x: bits(N), y: bits(N))
begin
    return;
end

func testUnforcedBits {M: integer {16,32,64}, L: integer {8,16,32}} (
    x: bits(M), y: bits(L)
    )
begin
    unforcedBits(x,y); // ILLEGAL
    // Both formals are of width N
    // But the widths of the actuals are M and L
    // Which are not necessarily equal to each other.
    // So one or other actual will not type satisfy the formal
    // after invocation width is applied

    unforcedBits(x as bits({16,32}), y); // ILLEGAL
    // This still does not make the widths of the actuals equal

    unforcedBits(x as bits({16,32}), y as bits({16,32}) ); // ILLEGAL
    // Nor does this

    unforcedBits(x, y as bits(M)); // Legal
    // The arguments now have the same width, but an execution-time check is
    // required to ensure that y meets the checked type conversion

    // The same applies to primitive operations:
    var z = x+(y as bits(M)); // Legal
end
```

The following example illustrates a more complex relationship between the formal arguments and the parameter `N`.

```
func unforcedBits {N} (x: bits(N*2), y: bits(N))
begin
    return;
end

func testUnforcedBits {L: integer{8,16,32}, M: integer {16,32,64}} (
     x: bits(M), y: bits(L)
    )
begin
    unforcedBits(x,y); // ILLEGAL
    // Formals are of width N*2 and N
    // But the widths of the actuals are M and L
    // and (M == L*2) is not necessarily true

    unforcedBits(x as bits(L*2), y); // Legal
    // The arguments now have the correct widths,
    // but an execution-time check is
    // required to ensure that x meets the checked type conversion

    // The same applies to primitive operations:
    var z = x as bits(L*2) + y; // Legal
end
```

7.8.3 Primitive Operators

ASL follows mathematical and programming language tradition of allowing operators such as $+$ to be overloaded to refer to one of several different operations. In the same way that subprograms are matched to the appropriate subprogram declarations (see 7.8.2 *Subprogram invocations*), primitive operators in expressions are matched to the appropriate operation.

For ease of reference within this manual, we give each such combination of operator and base types a unique name in Tables Table 7.1, Table 7.2, Table 7.3, Table 7.4, Table 7.5 and Table 7.6.

R_{BKNT} An operation *implements* an operator if it appears in the same row as the operator in one the tables in section 7.8.4 *Operator definitions*.

R_{JGWF} An expression which invokes a primitive operator *matches* an operation if the operation implements the operator and the operands of the expression *type-satisfy* the corresponding operands of the operation as shown in the tables in section 7.8.4 *Operator definitions*.

R_{TTGQ} It is a type-checking error if an expression which invokes a primitive operator does not match exactly one primitive operation.

7.8.4 Operator definitions

Table 7.1: Boolean operators

Operator	Operand 1	Operand 2	Result	Operation Name
`!`	boolean	-	boolean	not_bool
`&&`	boolean	boolean	boolean	and_bool
`||`	boolean	boolean	boolean	or_bool
`==`	boolean	boolean	boolean	eq_bool
`!=`	boolean	boolean	boolean	ne_bool
`-->`	boolean	boolean	boolean	implies_bool
`<->`	boolean	boolean	boolean	equiv_bool

Table 7.2: Bitvector operators

Operator	Operand 1	Operand 2	Result	Operation Name
`+`	bits(N)	bits(N)	bits(N)	add_bits
`+`	bits(N)	integer	bits(N)	add_bits_int
`-`	bits(N)	bits(N)	bits(N)	sub_bits

Operator	Operand 1	Operand 2	Result	Operation Name
`-`	bits(N)	integer	bits(N)	sub_bits_int
`NOT`	bits(N)	-	bits(N)	not_bits
`AND`	bits(N)	bits(N)	bits(N)	and_bits
`OR`	bits(N)	bits(N)	bits(N)	or_bits
`XOR`	bits(N)	bits(N)	bits(N)	xor_bits
`==`	bits(N)	bits(N)	boolean	eq_bits
`!=`	bits(N)	bits(N)	boolean	ne_bits

Table 7.3: Integer operators

Operator	Operand 1	Operand 2	Result	Operation Name
`-`	integer	-	integer	negate_int
`+`	integer	integer	integer	add_int
`-`	integer	integer	integer	sub_int
`*`	integer	integer	integer	mul_int
`^`	integer	integer	integer	exp_int
`<<`	integer	integer	integer	shiftleft_int
`>>`	integer	integer	integer	shiftright_int
`DIV`	integer	integer	integer	div_int
`DIVRM`	integer	integer	integer	fdiv_int
`MOD`	integer	integer	integer	frem_int
`==`	integer	integer	boolean	eq_int
`!=`	integer	integer	boolean	ne_int
`<=`	integer	integer	boolean	le_int
`<`	integer	integer	boolean	lt_int
`>`	integer	integer	boolean	gt_int
`>=`	integer	integer	boolean	ge_int

Table 7.4: Real operators

Operator	Operand 1	Operand 2	Result	Operation Name
`-`	real	-	real	negate_real
`+`	real	real	real	add_real

Operator	Operand 1	Operand 2	Result	Operation Name
`-`	real	real	real	sub_real
`*`	real	real	real	mul_real
`^`	real	integer	real	exp_real
`/`	real	real	real	div_real
`==`	real	real	boolean	eq_real
`!=`	real	real	boolean	ne_real
`<=`	real	real	boolean	le_real
`<`	real	real	boolean	lt_real
`>`	real	real	boolean	gt_real
`>=`	real	real	boolean	ge_real

Table 7.5: String operators

Operator	Operand 1	Operand 2	Result	Operation Name
`==`	string	string	boolean	eq_string
`!=`	string	string	boolean	ne_string

Table 7.6: Enumeration operators

Operator	Operand 1	Operand 2	Result	Operation Name
`==`	enumeration	enumeration	boolean	eq_enum
`!=`	enumeration	enumeration	boolean	ne_enum

7.8.5 Primitive operations on integers

See also *7.9 Conditional expressions*.

R_{ZYWY} If both operands of an integer binary primitive operator are integers and at least one of them is an unconstrained integer then the result shall be an unconstrained integer.

R_{BZKW} If both operands of an integer binary primitive operator are constrained integers and at least one of them is the under-constrained integer then the result shall be an under-constrained integer.

R_{KFYS} If both operands of an integer binary primitive operation are well-constrained integers, then it shall return a constrained integer whose constraint is calculated by applying the operation to all possible value pairs.

I_{YHRP} The calculation of constraints shall cause an error if necessary, for example where a division by zero occurs, etc.

```
func f(x: integer {2, 4}, y: integer {-1..1})
begin
    let z = x DIV y; // Illegal
    // type of z involves constraints with division by zero
    let ok = x DIV (y as {-1,1});
    // legal but incurs runtime check
end
```

I_{YHML} The constraint on the result of an integer binary primitive operation ‘a OP b’ where a is of type ‘integer {ka}’ and b is of type ‘integer {kb0, .., kbn}’ is ‘integer {ka OP kb0, .., ka OP kbn}’.

I_{VMZF} Note that implicit constraints allow compile-time-constant results of primitive operations to be well-constrained integers. See also *7.7.5 Implicit constraints for compile-time-constant integer expressions*.

I_{YXSY} If the result of a primitive operation which returns an unconstrained integer is used as a bitwidth then it will require a checked type conversion.

Example:

```
func primitiveConstraint(value: integer {8,16})
begin
    let j = value*(1+1);
    // (1+1) is a compile-time-constant expression
    // hence the type of (1+1) is integer {2}
    // and j is of type integer {16,32}

    var factor: integer = 2; // factor is of type integer
    // Note that without the explicit type, factor would be integer {2}

    let k = value * factor; // k is of type integer
    // since factor is unconstrained

    let c = if (factor==2) then value * 2 else value;
    // The 'then' subexpression is integer {16,32}
    // The 'else' subexpression is integer {8,16}
    // The conditional expression (and hence c)
    // is of type integer {8,16,32}
end
```

Note

Where a declaration has an implicit type, and the initialization expression is an implicitly constrained integer (see *7.7.5 Implicit constraints for compile-time-constant integer expressions*), the implicit type would previously have been `integer` but is now a well-constrained integer. This change may prove inconvenient for some code. For example, the declaration of `factor` in a later example below requires the `: integer` to avoid it being a constrained integer variable!

```
func implicitTypes()
begin
    var factor = 2;
    // *Mutable* storage elements of type integer {2} are not very useful...
    var c = if (factor==2) then i*2 else i;
    // but this might be an acceptable implicit type for c of integer {8,16,32}..
end
```

When declaring a mutable object, it is recommended to use an explicit type if there is any doubt over whether the initializing expressions’s type will have the correct domain of values that the object should have over its lifetime. A declaration of a mutable object without an initializer always requires an explicit type.

7.8.6 Primitive operations on bitvectors

R_{KXMR} If the operands of a primitive operation are bitvectors, the widths of the operands must be equivalent statically evaluable expressions.

I_{LGHJ} This rule means that when operating on two bitvectors of different types (e.g. a `bits(M)` and a `bits(N)`), a checked type conversion should be used to indicate that they are known to be of the same width. The intention is that, instead of silently inserting an execution-time check, an ASL processor will error by default without an explicit checked type conversion from the author.

Example

```
func check{M: integer{4,8}}(flag: boolean, x: bits(M), y: bits(8)) ==> boolean
begin
    if flag then
        return x == y;                // invalid; cannot determine x is bits(8)
        return (x AS bits(8)) == y;   // valid
    end
end
```

7.9 Conditional expressions

R_{XZVT} The type of a conditional expression is the lowest common ancestor (see 7.9.1 *Lowest common ancestor* below) of the types of the `then` and `else` expressions.

Note

It has been suggested that the type of a conditional expression whose `then` and `else` branches are different sized bitvectors could be a constrained type whose constraint is the union of the widths.

Note however that we have been trying to exclude such cases since they are a potential source of errors. For example, the following declaration of `conditionalFun` requires checked type conversions on the branches to ensure the return type is satisfied:

```
func conditionalFun(N: integer {1,2}, argT: bits(1), argF: bits(2)) => bits(N)
begin
    if (N==1) then
        return argT as bits(N);
    else
        return argF as bits(N);
    end
end
```

By the same principle of excluding unintentional errors, this example with a conditional expression in place of the conditional statement should also have checked type conversions:

```
func conditionalFun(N: integer {1,2}, argT: bits(1), argF: bits(2)) => bits(N)
begin
    return (if (N==1) then argT as bits(N) else argF as bits(N));
end
```

For constrained integers the situation is different:

```
func conditionalFun(N: integer {1,2}, argT: integer{1}, argF: integer{2}) => integer {1,2}
begin
    if (N==1) then
        return argT;
    else
        return argF;
    end
    // The return expression integer {1,2} satisfies the return type
    // No checked type conversion is required.
end
```

Hence we reject the suggestion of allowing conditional expressions to have different bitvector widths in true and false branches.

7.9.1 Lowest common ancestor

R_{YZHM} The lowest common ancestor of types S and T is:

- If S and T are the same type: S (or T).
- If S and T are both named types: the (unique) common supertype of S and T that is a subtype of all other common supertypes of S and T.
- If S and T both have the structure of array types with the same index type and the same element types:
  - If S is a named type and T is an anonymous type: S
  - If S is an anonymous type and T is a named type: T
- If S and T both have the structure of tuple types with the same number of elements and the types of elements of S type-satisfy the types of the elements of T and vice-versa:
  - If S is a named type and T is an anonymous type: S

  - If S is an anonymous type and T is a named type: T
  - If S and T are both anonymous types: the tuple type with the type of each element the lowest common ancestor of the types of the corresponding elements of S and T.
- If S and T both have the structure of well-constrained integer types:
  - If S is a named type and T is an anonymous type: S
  - If T is an anonymous type and S is a named type: T
  - If S and T are both anonymous types: the well-constrained integer type with domain the union of the domains of S and T.
- If either S or T have the structure of an unconstrained integer type:
  - If S is a named type with the structure of an unconstrained integer type and T is an anonymous type: S
  - If T is an anonymous type and S is a named type with the structure of an unconstrained integer type: T
  - If S and T are both anonymous types: the unconstrained integer type.
- If either S or T have the structure of an under-constrained integer type: the under-constrained integer type.
- Else: undefined.

7.10 Comparison operations

R$_{\texttt{SQXN}}$ If an argument of a comparison operation is a constrained integer then it is treated as an unconstrained integer.

R$_{\texttt{MRHT}}$ If the arguments of a comparison operation are bitvectors then they must have the same determined width.

EXAMPLE

```
func compare(int1: integer {1,2},
             int2: integer {4,5},
             bit1: bits(int1),
             bit2: bits(int2),
             int3: integer {1,2},
             bit3: bits(int3)
             )
begin
    var cond: boolean;
    cond = int1 == int2; // Legal
    cond = bit1 == bit2; // Illegal
    cond = bit1 == bit3; // Illegal
    // Type check failure since type checker knows
    // int1 ==> int1
    // int3 ==> int3
    // and cannot prove they are always the same
end
```

7.11 Bitvector concatenation

7.11.1 Determined width of a bitvector concatenation

$\mathrm{R}_{\mathtt{NYNK}}$ If any argument of a bitvector concatenation has undetermined width then the result of the bitvector concatenation has undetermined width.

$\mathrm{R}_{\mathtt{KCZS}}$ If all arguments of a bitvector concatenation have determined width then the result of the bitvector concatenation is a determined width bitvector whose width is the sum of the arguments' widths.

Example

```
func f {N} (a: bits(N), i: integer {1..N-1})
begin
    // a is an under-constrained bitvector of determined width `N`

    let b: bits(N) = [a[i-1:0], // constrained of determined width (i)
                      a[N-1:i]  // constrained of determined width (N-i)
                     ];         // result is constrained of determined width N
end
```

7.11.2 Constraint of a bitvector concatenation

$\mathrm{R}_{\mathtt{FHYZ}}$ If at least one argument is an under-constrained width bitvector then the result of the bitvector concatenation is an under-constrained width bitvector.

$\mathrm{R}_{\mathtt{VBMX}}$ If all arguments of bitvector concatenation are fixed width bitvectors then the result of the bitvector concatenation is a fixed width bitvector.

$\mathrm{R}_{\mathtt{XVWK}}$ If no arguments of a bitvector concatenation with `n` arguments are under-constrained width bitvectors and at least one argument is a constrained width bitvector then the result of the concatenation is a constrained width bitvector whose constraint is C which is calculated as follows:

- if the i'th argument is a constrained width bitvector, then the set C_i is the constraint of the argument's width
- if the i'th argument is a fixed width bitvector, then C_i is the set containing only the argument's width
- the set C is $\{x | y_1 \in C_1 \wedge \cdots \wedge y_n \in C_n \wedge x = \sum_{i=1}^{n} y_i\}$

Examples

```
func f(P: integer {2,4,8})
begin

    var opA = [Zeros(P-1), '1'];
    // First element of concatenation is a constrained width bitvector of determined width
    // with constraint {1,3,7} and width (P-1)
    // Second element of concatenation is a fixed width bitvector of width 1
    // For the purpose of calculating the resulting type, it has constraint {1}
    // The result is a constrained width bitvector of determined width
    // with constraint {1+1,3+1,7+1} and width ((P-1)+1)

    // Hence the type of opA is bits(P as {2,4,8})
end
```

7.12 Bitslices

$\mathrm{I}_{\mathrm{MJWM}}$ The result of a bitslice operation is a bitvector of determined width. This follows from the requirement that the width of a bitslice must be any non-negative, statically evaluable integer expression.

7.13 Checked type conversions

I_{XVBG} Due to the dynamic structure of a specification it may be that, during execution, the value of an expression is within a specific domain. However, the static type checker may not be able to infer this since it does not perform detailed analysis of dynamic behavior. In these situations, an execution-time check can be explicitly indicated by the author using a checked type conversion.

For example, a check that a bitvector expression has the correct width may be inserted as in the following example:

```
func widCheck {M} (N: integer, b: bits(M)) => bits(N)
begin
    if (N==M) then
        return b as bits(N); // Inserts execution-time check
    else
        return Zeros(N);
    end
end
```

Without that checked type conversion, the return statement would fail type checking since it is not known to the type checker that the `bits(M)` type of `b` satisfies the return type `bits(N)`.

The check may be elided if the compiler can determine that it will always be true. In the above example, a smart compiler will see that the immutable values `N` and `M` are equal when the `return b` is executed.

R_{GYJZ} A checked type conversion which is applied to a bitvector has the same width as the bitvector.

I_{SZVF} By definition, a checked type conversion where the required type is a bitvector of determined width must have the same width as the bitvector it is applied to. Additionally, in the case of a checked type conversion where the required type is a bitvector of undetermined width, the result of the checked type conversion will have the width of the bitvector it is applied to.

7.13.1 Checked type conversions on expressions

R_{PZZJ}

- If type-checking determines that the expression type-satisfies the required type, then no further check is required.
- If the expression only fails to type-satisfy the required type because the domain of its type is not a subset of the domain of the required type, an execution-time check that the expression evaluates to a value in the domain of the required type is required.

I_{VQLX} The checked type conversion rule permits an expression to be used if is subtype compatible with the required type at execution-time.

R_{YCPX} An execution-time check for a checked type conversion should not be failed before the expression is evaluated. For example, the check on `y` below should not cause an execution-time error if the invocation of f1 returns `FALSE` at execution time:

```
func checkY (y: integer)
begin
    if (f1() && f2(y as {2,4,8})) then pass; end
end
```

I_{ZLBW} The use of a checked type conversion allows uses of values which would normally be forbidden by the type system. As such it may result in an execution-time check being inserted. Since tools may elide unnecessary execution-time checks, it may be useful for tools to offer an optional mode where non-elided execution-time checks cause a warning to be issued so that authors may review complex checked type conversions manually.

7.13.1.1 Checked type conversions on integer expressions

I_{TCST} The rule about domains in the definitions of subtype-satisfaction and type-satisfaction means that it is illegal to use the unconstrained integer where a constrained integer is expected. A checked type conversion can be used to overcome this.

I_{CGRH} A checked type conversion allows code to explicitly mark places where uses of constrained types would otherwise cause a type-check error. The intent is to reduce the incidence of unintended errors by making such uses fail type-checking unless the checked type conversion is provided.

I_{YJBB} Primitive operations may return unconstrained integers, so a checked type conversion may be required on primitive operation expressions. See also *7.8.5 Primitive operations on integers*.

Examples

```
func CTCs(x: integer)
begin
    var A: bit;

    let B: integer = A as integer; // ILLEGAL
    // bit cannot be an integer

    let C: integer {8,16} = x as {8,16};
    // The execution-time check is `x IN {8,16}`

    let D: integer {8,16} = C as {8,16};
    // Type checker can determine that C is already an integer {8,16}
    // so no further check is required
end
```

7.13.2 Examples of checked type conversions

7.13.2.1 Checked type conversion with configs

I_{GQRD} The following example shows the declaration of a function `getDefault` whose return type is a bitvector of undetermined width which is not dependent on a parameter. Since it is dependent on a `config` value, the declaration is legal - there is no overloading of the result type.

```
config systemWid: integer {8,16} = 8;
// systemWid==>systemWid

func getDefault() => bits(systemWid)
begin
    var ret: bits(systemWid) = Zeros(systemWid);
    return ret;
end

func systemOp(x: bits(systemWid)) => bits(systemWid)
begin
    return (NOT x);
end

func getWidth {N} (x: bits(N)) => integer
begin
    return N;
end

func test()
begin
    var addr = getDefault();
    // Type of addr is bits(systemWid)
    // which is a constrained width bitvector bits({8,16})

    addr = systemOp(addr); // Legal
    // Both LHS and RHS are bits(systemWid)

    let N = getWidth(addr) as {8,16};
    // Invokes execution-time check that RHS is IN {8,16}
    // Type of N is integer {8,16}
    // N==>N

    var newAddr: bits(N) = [ Zeros(N-1), '1'];
    newAddr = systemOp(addr); // ILLEGAL
    // LHS is bits(N) but RHS is bits(systemWid)

    newAddr = systemOp(addr) as bits(N);
    // Incurs an execution-time check that (N==systemWid)

    - = newAddr + addr; // ILLEGAL
    // Type checker cannot demonstrate (systemWid==N)

    newAddr = newAddr + (addr as bits(N)); // Legal
    // Incurs an execution-time check that (N==systemWid)

    addr = newAddr; // ILLEGAL
    addr = newAddr as bits(systemWid); // Legal
end
```

I_{GJZQ} The following example illustrates a simple use of constrained types with `configs`.

```
// a global which may only hold the values 8, 16 or 32
var gWid: integer {8,16,32};

// a global constant which holds the value 64 and has type integer {64}
constant constantValue = 64;

// The initializer for the following config is a compile-time-constant
// whose value is 32 hence its type is integer {32} which type checks
// against the type of configWid
config configWid: integer {8,16,32} = constantValue DIV 2;
// Although constantValue is of type integer, it is compile-time-constant
// so the initializer of configWid is `32: integer{32}`
config halfWid: integer {4,8,16} = configWid DIV 2;
```

7.13.2.2 Checked type conversion on literal

I_{MKPR} The following example shows that even literals may require a checked type conversion to ensure they meet type-satisfaction requirements.

```
func invokedN {N: integer {8,16,32}} (x: bits(N)) => bits(N)
begin
    var myBits: bits(N);
    // The type of myBits is bits(N as {8,16,32})
    if N == 8 then // "GUARD"
        myBits = '10101010'; // ILLEGAL
        // since the type of myBits is not of width 8

        myBits = '10101010' as bits(N); // Legal  (line AS01)
        // type checker inserts execution-time check `N==8`
    else
        myBits = Zeros(N);
    end
    return myBits;
end

func test(M: integer)
begin
    var myVal: bits(M as {16,32});
    // Incurs execution-time check that (M IN {16,32})
    var myResult: bits(M as {8,16});
    // Incurs execution-time check that (M IN {8,16})

    myResult = invokedN(myVal); // ILLEGAL
    // The return type of invokedN(myVal) is bits(M as {16,32})
    // which does not type satisfy myResult

    myResult = invokedN(myVal as bits({8,16})); // returns a bits({8,16})
    // Execution-time check that M IN {8,16}
    // Note that the only value of M which can cause this invocation is `16`
    // so the execution-time check `N==8` in invokedN at "AS01" is not executed
    // due to its `if N==8` above failing

    myVal = invokedN(myResult) as bits({16, 32});
    // Execution-time check that the returned value is IN {16,32}
    // Invocation matches signature so it type-checks
    // Note that myResult may be a bits(8),
    // so the check at line AS01 may be executed
end
```

7.13.2.3 Checked type conversion with `bits({...})`

I_BHLN In the following example, the actual argument of the invocation of `myFunc` on the last line has a checked type conversion which does not supply a width. This results in the actual argument being treated as a bitvector of determined width equal to the determined width of `myVal`, but with a different constraint from that of `myVal`, namely `{8,16,32}` instead of `{8,16,32,64}`. See 7.13 *Checked type conversions* for details of the width of a checked type conversion which is applied to a bitvector.

The type checker knows that the invocation of `myFunc` returns a value whose width is the same as the width of the actual argument. However, the constraint `{8,16,32,64}` on `myVal` alone would not satisfy the constraint `8,16,32` on the formal argument `myInput` hence the checked type conversion is required.

```
func myFunc {N: integer{8,16,32}} (myInput: bits(N)) => bits(N)
begin
    return Zeros(N);
end

func MyVectorInstruction()
begin
    let myWid: integer {8,16,32,64} = f();
    var myVal: bits(myWid) = g();
    if myWid == 64 then
        myVal[31:0]  = myFunc(myVal[31:0]);
        myVal[63:32] = myFunc(myVal[63:32]);
    else // author knows myVal is not bits(64)
        myVal = myFunc(myVal as bits({8,16,32}));
    end
end
```

7.14 Bitvector width comparison

In order to ensure that type checking can compare the widths of bitvector expressions during compilation, bitvector widths are always statically evaluable expressions. See rules YTGH, BDJK, WZCS and RMTQ.

R_{XNBN} When determining the widths of bitvectors, the type checker shall make use of statically evaluable expressions.

I_{GQYG} This means that `constant` and `let` identifiers can be used to demonstrate that a bitvector's width is acceptable.

Comparing bitvector widths involves maintaining a mapping from immutable identifiers to their values. Where the value of an immutable identifier is not given as a statically evaluable expression, the mapping should simply map the identifier to itself.

Comparisons of bitvector width requires the ability to use these mappings to reduce immutable terms in an expression using arithmetic rewrites and normalization.

Examples:

The following example code is annotated with comments describing the progress of a type checker, using `<x>--><e>` to indicate that an occurrence of `<x>` can be rewritten to `<e>`.

```
var gInt: integer {1,2,3}; // a constrained mutable global

func mutables(wid: integer)
begin
    // type checker knows wid-->wid

    constant mod = 1;
    // RHS is immutable so mod-->1

    let size01 = wid + gInt;
    // RHS is mutable so size01-->size01

    var data01: bits(size01+1);
    // size01 reduces to size01 so
    // type checker knows data01 is (size01+1) wide

    let size02 = wid + gInt + mod;
    // RHS is mutable so size02-->size02

    var data02: bits(size02);
    // size02-->size02 so
    // type checker knows data02 is (size02) wide

    data01=data02;
    // type checker emits an error "Widths do not match"
    // since it cannot tell that (size01+1)==(size02)
end


func immutables(wid: integer)
begin
    // type checker knows wid-->wid

    constant mod = 1;
    // RHS is immutable so mod-->1

    let size01 = wid;
    // RHS is immutable so size01-->wid

    var data01: bits(size01+1);
    // size01-->wid so
    // type checker knows data01 is (wid+1) wide

    let size02 = wid + mod;
    // RHS is statically evaluable
    // and mod-->1 so
    // type checker knows size02-->(wid+1)
    var data02: bits(size02);
    // size02-->(wid+1) so
    // type checker knows data02 is (wid+1) wide

    data01=data02;
    // type checker knows that (wid+1)==(wid+1)
    // Widths match
end
```

Since the type-checker uses statically evaluable expressions when determining bitvector widths, immutable values of type `integer` are treated as *width constraints* by the type-checker and used during width inference.

For example, the declaration `let F: integer = (N - E)- 1;` in the following function is used as a width constraint

This is a minor, but important, extension to allow the floating point functions to be written in a more natural style. Consider the following function and how we can type-check the final return statement.

```
func FPZero(sign: bit, N: integer {16,32,64}) => bits(N)
begin
    // type checker knows N-->N
    let E: integer = if N == 16 then 5 elsif N == 32 then 8 else 11;
    // type checker knows E-->if N == 16 then 5 elsif N == 32 then 8 else 11
    let F: integer = (N - E) - 1;
    // type checker knows F-->N-(if N == 16 then  5 elsif N == 32 then  8 else    11) - 1
    // which is F-->(if N == 16 then (N -5) elsif N == 32 then ( N-8) else (N-11)) - 1
    // which is F-->(if N == 16 then (16-5) elsif N == 32 then (32-8) else (N-11)) - 1
    // which is F-->(if N == 16 then (  11) elsif N == 32 then (  24) else (N-11)) - 1
    // which is F-->(if N == 16 then (  10) elsif N == 32 then (  23) else (N-12)) - 1
    var exp  = Zeros(E);
    var frac = Zeros(F);
    return [sign, exp, frac];
    // type checker knows width of return expression is 1 + E + F
    // which is 1
    //    + (if N == 16 then  5 elsif N == 32 then  8 else    11 )
    //    + (if N == 16 then 10 elsif N == 32 then 23 else (N-12))
    // which is 1
    //    + (if N == 16 then 15 elsif N == 32 then 31 else (N- 1))
    // which is
    //      (if N == 16 then 16 elsif N == 32 then 32 else     N )
    // which is N
end
```

7.15 Base values

Each type has a *base value*, used in 6.2.2 *Initialization of locals* if an initializer is not supplied.

7.15.1 Base value of integers

R$_{NJDZ}$ The base value of the unconstrained integer is `0`.

R$_{CFTD}$ The base value of a well-constrained integer is the closest value to zero in its domain. If the closest positive and negative value are equally close, the positive value is used.

R$_{QGGH}$ There is no base value for the under-constrained integer.

I$_{WVQZ}$ Since only subprogram parameters can be under-constrained integers, and since they will be initialized by their invocation, there is no need to have a base value for them.

7.15.2 Base values of other types

R$_{GYCG}$ The base value of the `real` type is `0.0`.

R$_{WKCY}$ The base value of the `string` type is the empty string.

R$_{LCCN}$ The base value of an `enumeration` type is the first (i.e. leftmost) enumeration literal in the declaration of that enumeration type.

```
type SigEnum of enumeration {LOW, HIGH};  // SigEnum has base value LOW
```

R$_{CPCK}$ The base value of a boolean is `FALSE`.

R$_{ZVPT}$ The base value of `bits(N)` is the N-bit bitvector containing only zeros.

I$_{QFZH}$ For example the base value of `bits(2)` is `'00'`.

R$_{WGVR}$ The base value of an array type is an array with the same element indices as the array type, whose elements have the base value of the array's element type.

I$_{PGSS}$ For example the base value of an `array [2] of integer` is an `array [2] of integer` in which both elements have the value `0`.

R$_{QWSQ}$ The base value of a tuple type is a tuple whose elements have the base values of their types.

I$_{HMRK}$ For example, the base value of a tuple `(integer, boolean)` is the tuple `(0, FALSE)`.

R$_{MBRM}$ The base value of a record type is a record whose elements have the base values of their types.

Example:

Given the following declaration of a record type `a_record_ty`:

```
type a_record_ty of record {
    flag : boolean,
    count: integer,
    data : bit
};
```

declaring a local variable `a_record` of that type, with no initializer, is equivalent to the following where `a_record` is initialized with the base value of `a_record_ty`:

```
    var a_record = a_record_ty {
        flag = FALSE,
        count = 0,
        data = '0'
    };
```

R$_{SVJB}$ The base value* of an exception type is an exception whose elements have the base values of their types.

7.16 Type checking examples

7.16.1 Named types example

$\mathrm{X}_{\mathrm{XSWL}}$ The intent is that, by default, any named type should not be assignable to or from any other named type, even if they coincidentally have the same structure. Where assignment between named types is desired it is usually achieved by grouping related types via a common supertype. For example, given the following code:

```
// Neither ADDR nor PHYSICAL_ADDR is a subtype of the other.
type ADDR of bits (32) {};
type PHYSICAL_ADDR of ADDR;

var addr    : ADDR;
var physical: PHYSICAL_ADDR;

// For the function "raw_addr",

func raw_addr(x: ADDR) => bits(32)
begin
    // x may be used as the expression in the return statement
    // since the return type is type satisfied by the type of x
    return x;
end

func raw_physical_addr(x: PHYSICAL_ADDR) => bits(32)
begin
    return x;
end
```

- The assignments `addr = physical;` and `physical = addr;` are illegal.
- The following assignments are legal:

```
func addresses()
begin
    var tmp: bits(32);
    // primitive type bits(32) is type-satisfied by both ADDR and PHYSICAL_ADDR

    tmp      = addr;
    physical = tmp;
    tmp      = ['0', tmp[30:0]];
    addr     = tmp;

    physical = raw_addr(addr);
    addr     = raw_physical_addr(physical);

    physical = addr[31:0]; // a bitslice is of type bits(N)
    addr     = physical[31:0];
end
```

- This example demonstrates how a constant can be given a named type.

```
type Char of integer{0..255};
type Byte of integer{0..255};

constant K: Char = 210;

var c: Char;
var b: Byte;

function f()
begin
    c = 210;  // legal: c has the structure of integer and can be assigned an integer
    c = K;    // legal: K has type Char and can be assigned to a Char
    b = K;    // illegal: a Char cannot be directly assigned to a Byte
end
```

7.16.2 Anonymous types example

I_{SBCK} This example illustrates the use of anonymous types as permitted by the type-satisfaction rule.

```
type T1 of integer;           // the named type `T1` whose structure is integer
type T2 of integer;           // the named type `T2` whose structure is integer
type pairT of (integer, T1); // the named type `pairT` whose structure is (integer, integer)

func tsub01()
begin
    var dataT1: T1;

    var pair: pairT = (1,dataT1);
    // legal since right hand side has anonymous, non-primitive type (integer, T1)

    let dataAsInt: integer = dataT1;
    pair = (1, dataAsInt);
    // legal since right hand side has anonymous, primitive type (integer, integer)

    let dataT2: T2 = 10;
    pair = (1, dataT2);
    // illegal since right hand side has anonymous, non-primitive type (integer, T2)
    // which does not subtype-satisfy named type pairT
end
```

7.16.3 Constrained types examples

G_{MTXX} It shall be an error to use a value of type <T> where a value of constrained type <S> is required if <T> can hold a value which <S> cannot.

I_{KJDR} Due to the various type-checking rules, we allow <T> to be used where an <S> is expected if <T> type satisfies <S>. This includes in assignments, function arguments and constraints. We already know that if <T> type satisfies <S> then the domain of <T> must be a subset of the domain of <S>, which corresponds to the requirement MTXX above.

Note that although it may seem desirable for the rules for legal assignments to be the same as those for matching actuals to formals in invocations, there is a fundamental difference in the use of these two features.

- An assignment is to a storage element which must have a specific width during execution, so the compiler must be satisfied that the RHS value is of the correct width.
- An invocation matches against formal arguments which have no specific width but may match multiple widths.

For example:

```
func invokeMe {N: integer {8,16,32}} (x: bits(N))
begin
    return;
end

func test(M: integer {8,16,32}, L: integer {8,16})
begin
    var myM: bits(M);
    var myL: bits(L);

    if (M != L) then
        return;
    end
    // Note the type-checker does not do full program analysis
    // So it does not know that M==L after this statement

    myM = myL; // ILLEGAL
    // myM and myL are constrained width bitvectors of determined widths
    // M and L respectively.
    // The type-checker does not know (M==L), so subtype-satisfaction
    // disallows this use of myL.
    myM = myL as bits(M); // Legal
    // The author explicitly claimed that myL has the width of myM
    // An execution-time check of (M==L) is required

    invokeMe(myL); // Legal
    // The parameter N is taken to be the value which corresponds
    // with the width of myL and the width of myL is an integer {8,16}
    // which complies with the declaration of parameter 'N'
    // The rules for subtype-satisfaction are satisfied since
    // the formal 'x' and the actual 'myL' are of the same determined width.
end
```

7.16.3.1 Example: constrained integer types

```
type Ity of integer {2,4,8};

func tsub02()
begin
    var A: integer {2,4,8};
    var B: integer {2,4};
    // A and B have anonymous types

    A = B; // legal: FMXK clause 2
    B = A; // illegal: domain of A's type is not a subset of domain of B's type

    var I: Ity;
    I = A; // legal: FMXK clause 2
    I = B; // legal: FMXK clause 2

    B = I; // illegal: subtype-satisfaction fails due to domains
    A = I; // legal: FMXK clause 2
end
```

```
var gInt: integer; // unconstrained global integer

func f1()
begin
    var myInt : integer         = gInt;    // Legal

    var myIntA: integer {1..10} = myInt as integer {1..10};
    // Legal: incurs execution-time check that (myInt IN {1..10})

    var myIntB: integer {0..20} = myIntA;
    // Legal: type satisfaction due to domains (no execution-time check required)

    myIntA = myIntB;
    // Type check fail even if smart compiler believes myIntB holds
    // a value from myInt_A since
    // `integer {0..20}` does not type satisfy
    // `integer {1..10}` due to domains
end
```

```
func wid1() => integer {8,16}
begin
    return someWid1;
end

func wid2() => integer {4,8}
begin
    return someWid2;
end

func f2()
begin
    let w1: integer {2,4,8,16} = wid1();
    // RHS is not statically evaluable so the only thing the type
    // checker can deduce is that w1==>w1
    // We do not constrain w1 based on the return type of wid1
    // since this may be intentional to avoid checked type conversions later
    // e.g. to ensure b1 has the correct type.

    let w2: integer {4,8,16}    = wid2();
    // RHS is not statically evaluable so w2==>w2

    // The set of possible widths of a bitvector must be statically evaluable.
    // All of the following are:
    var b1: bits(w1); // type is bits(w1 as {2,4,8,16})
    var b2: bits(w2); // type is bits(w2 as {4,8,16})

    b1 = b2; // Type check fail
    // Type checker cannot determine w1==w2
    //so we require a Checked Type Conversion:
    b1 = b2 as bits(w1); // Type check PASS
    // but requires an execution-time width check that (w2==w1)
end
```

Note

The example above shows a `let w1` declaration which is not constrained by its RHS initializer. For `var` declarations this is required since the `var` may later be assigned a value outside the initializers domain:

```
func varConstraint()
begin
    var a: integer {1..100} = 1;
    a = 2;
end
```

It has been proposed that for `let` bindings we might constrain the type (i.e. `w1` would be `integer {8,16}`) since the value cannot change and the type checker might flag more problems with subsequent code.

However, for now it seems that it is better to leave `let` as the same as `var` for easier readability and to help with code maintenance, for example, given the code:

```
func bar() => integer {8,16,32}
begin
    return 8;
end

func foo()
begin
   let myVal: integer {8,16,32} = bar();
end
```

future changes which expand the constraint of `bar` would be noticed since the declaration of `myVal` would become illegal.

Also, although it is not clear why you would write code like `w1` above, it may be that there is some documentation related reason for a specification to make such a statement about a `let` type. (Note that these types may be named types which carry configured constraint information. . .)

See also 4.4 *Global storage elements* for the contrasting behavior of bitvector initialization expressions.

7.16.3.2 More Invocation Examples

```
func bus {wid} (arg0: bits(wid), arg1: bits(wid*2)) => bits(wid)
begin
    // When type-checking the declaration of func bus
    // arg0 and arg1 are under-constrained width bitvectors
    // of determined width `wid`
    // Since wid is not a formal, it takes its value in an invocation from
    // the width of one of the the corresponding actuals

    return arg0;
end

// ------------------------------------------------------------
// Cases for invocation of function `bus`
// which has an under-constrained width bitvector formal

func legal_fun_fixed_width_actual () => bits(8)
begin
    let x: bits(8)  = Zeros(8);
    let y: bits(16) = Zeros(16);
    // bus's wid parameter takes its value as `8`
    // The invocation width of bus's arg0 is therefore `8`
    // x type satisfies arg0: bits(8)
    // The invocation width of bus's arg1 is therefore `8*2`
    // y type satisfies arg0: bits(16)
    return bus(x, y);
end

func legal_fun_underconstrained_actual (N: integer) => bits(N)
begin
    // N is a parameter, therefore it is an under-constrained integer
    var x: bits(N);
    var y: bits(N*2);
    // bus's wid parameter takes its value from the width of x
    // which is `N` which is an under-constrained integer
    // Therefore the type of arg0 with the invocation width `N` is
    // the under-constrained width bitvector of determined width `N`
    // which is type satisfied by x
    return bus(x, y);
end

func legal_fun_constrained_actual (arg: bits({32,64})) => bits(32)
begin
    // This invocation is OK because the actual has undetermined width
    // so the formal is treated as having undetermined width
    // and the domain of bits({32,64}) is a subset of the domain of the
    // undetermined width bitvector
    return bus(arg, [arg,arg])[31:0];
end

func illegal_fun_parameter_mismatch (N: integer{32,64}, M: integer{64,128})
begin
    var argN: bits(N);
    var argM: bits(M);
    // Illegal invocation:
    // Either bus's wid takes its value from argN
    // in which case argM does not type satisfy arg1
    // OR bus's wid takes its value from argM
    // in which case argN does not type satisfy arg0
    let illegal = bus(argN, argM);
    // A checked type conversion might be useful...
    let legal = bus(argN, argM as bits(N*2));
end
```

7.16.3.3 More assignment examples

```
func assignBits {
    N:integer, M: integer
    } (
    someWid: integer {32,64},
    argN: bits(N), argM: bits(M)
    )
begin
```

```
    // argN and argM are immutable under-constrained width bitvectors
    // assignments to them are illegal

    // legal since widths and domains match
    var eightBits: bits(8) = Zeros(8);

    // legal: someBits has undetermined width so we require RHS to be a
    // bitvector whose domain is a subset of {32,64} which it is by the
    // declaration of someWid
    var someBits: bits({32,64}) = Zeros(someWid);


    // underconstrainedBits is a mutable under-constrained width bitvector
    // it can be assigned to
    var underconstrainedBits: bits(N);

    // underconstrainedBits has determined width `N`, so RHS must have same width
    underconstrainedBits = argN;      // legal since widths match
                                      // and domains are identical

    underconstrainedBits = argM;      // illegal since widths do not match
    underconstrainedBits = eightBits; // illegal since widths do not match
    underconstrainedBits = someBits;  // illegal since widths do not match
                                      // (someWid==N may be false)


    eightBits = underconstrainedBits; // illegal since widths do not match
    someBits  = underconstrainedBits; // illegal since widths do not match
                                      // (someWid==N may be false)
end
```

Chapter 8
Interpretation

This section is an informal description of how ASL is interpreted. It is expected to be replaced by a Semantics Reference Document in due course. If there is a conflict between the behavior described in this document and the Semantics Reference Document, the Semantics Reference Document will be considered the correct version.

8.1 Mutability

D_{FXST} Storage elements and identifiers which denote storage elements are either *mutable* or *immutable*.

See 4.4 *Global storage elements* and 6.2 *Local storage elements* for rules about which declarations create mutable and immutable storage elements.

R_{WDGQ} Immutable values may not be assigned to or otherwise have their values modified after initialization.

R_{ZDKC} The type of a mutable storage element shall not be an under-constrained integer.

Note

In order to maintain simple rules for a compiler to determine the range of values which an under-constrained integer may hold, we forbid mutable under-constrained integers since these would require the compiler to interpret every assignment to that storage element.

8.1.1 Statically evaluable expressions

I_{PKXK} We extend the concept of mutability to expressions such that we describe an expression as *statically evaluable* if its evaluation only involves the use of immutable values.

D_{YYDW} Any expression consisting solely of an immutable storage element or a [literal constant]{ 5.8 *Literal constants*} is a statically evaluable expression.

D_{CWVH} A compile-time-constant expression is a statically evaluable expression.

D_{HLQC} Any expression consisting of a primitive operation on statically evaluable operands is a statically evaluable expression

I_{LZCX} Note that this means a statically evaluable expression must not contain non-compile-time-constant getter or function invocations.

8.1.2 Deciding equivalence for statically evaluable expressions

I_{LHLR} The interpretation of ASL requires checking statically evaluable integer expressions for equivalence.

R_{RFQP} Two statically evaluable expressions being checked for equivalence are both reduced to canonical form, and then the two canonical forms are compared structurally.

R_{VNKT} A statically evaluable expression that must be checked for equivalence may contain an integer divide operation (`DIV`, but not `DIVRM`) only if the divisor is a product of non-zero compile-time-constant integer expressions and immutable integer variables.

Informally, reduction to canonical form involves:

- Recursively substituting immutable variable identifiers with their initialization expression, provided the initialization expression is a statically evaluable expression.
- Applying addition, subtraction, unary negation and exact division (`DIV`).
- Distributing addends across multiplication.
- Removing terms with a zero factor.
- Cancelling common factors.
- Sorting addends and factors by the immutable variable identifiers they contain, according to some common total order.

See 7.14 *Bitvector width comparison*.

8.2 Interpretation of functions and procedures

An ASL specification describes behavior in terms of the execution of a single thread of control. Executing a function or procedure may modify global state and can result in one of the following situations:

- The function/procedure returns successfully (i.e., without throwing an ASL exception or detecting a dynamic error). In this case, the result is a possibly modified global state and, for functions, a return value.
- The function/procedure throws an ASL exception. In this case, the result is a possibly modified global state and an exception object (that could be caught by the caller in a try-catch block, if desired).
- The function/procedure detects a dynamic error condition. This indicates an error in the specification and the global state and any other behavior is meaningless.
- The function/procedure enters an infinite loop. This indicates an error in the specification and the global state and any other behavior is meaningless.

Note

- Arm's implementation of ASL also supports some builtin functions for printing to the console, etc. These internal extensions are not used in Arm's published specifications but could be added to the above description in the usual manner.
- We note that, even in the simple case that a procedure returns successfully, the behavior is not a pure mathematical function from input states to output states. ASL provides explicit mechanisms for underspecification (e.g., `UNKNOWN`) and so the behavior of this simple case is better modeled as a relation between states.
- An alternative, more flexible, way of defining the meaning of ASL would be in terms of traces of interactions with external components such as the memory system or interrupt controllers and of reads and writes of the global state of the specification. In such a definition, the meaning of a function would be defined as a set of possible traces.

8.2.1 Execution-time subprograms

$\mathtt{I}_{\mathtt{NXJR}}$ Subprogram declarations in ASL are either *execution-time* subprogram declarations or *non-execution-time* subprogram declarations.

$\mathtt{D}_{\mathtt{CSFT}}$ A subprogram declaration is an execution-time declaration if it makes use of any of the following:

- an execution-time storage element
- an execution-time expression
- an execution-time subprogram invocation

$\mathtt{I}_{\mathtt{HYBT}}$ Subprogram invocations are also either *execution-time* or *non-execution-time* invocations.

$\mathtt{D}_{\mathtt{CCTY}}$ A subprogram invocation is an execution-time invocation if the invoked subprogram has an execution-time declaration or if the invocation contains any of the following:

- an execution-time storage element
- an execution-time expression
- a bitvector whose width is an execution-time expression

$\mathtt{I}_{\mathtt{LYKD}}$ Declaration CCTY means that an execution-time subprogram declaration shall only have execution-time invocations.

8.2.2 Compile-time-constant subprograms

I

$\mathrm{I}_{\mathrm{ZPWM}}$ Subprogram declarations in ASL are either *compile-time-constant* subprogram declarations or *non-compile-time-constant* subprogram declarations.

$\mathrm{D}_{\mathrm{KCKX}}$ A subprogram declaration is a compile-time-constant declaration if all of the following are true:

- the subprogram is side-effect-free
- all assignments in the subprogram are to the subprogram's local variables
- all expressions in the subprogram are compile-time-constant expressions
- any subprogram invocations made in the subprogram are compile-time-constant subprogram invocations

$\mathrm{I}_{\mathrm{NTYZ}}$ Standard (built-in) functions, as defined in Chapter 9 *Standard library*, are generally compile-time-constant, unless otherwise stated.

$\mathrm{I}_{\mathrm{MSZT}}$ Subprogram invocations are also either *compile-time-constant* subprogram invocations or *non-compile-time-constant* subprogram invocations.

$\mathrm{D}_{\mathrm{QNHM}}$ A subprogram invocation is a compile-time-constant invocation if all the following hold:

- the invoked subprogram is a compile-time-constant subprogram
- all of the actual arguments are compile-time-constant expressions
- all actual arguments which are bitvectors were declared with a constant expression width

Note

Subprogram declarations or invocations may be both non-compile-time-constant and non-execution-time.

For example, storage elements declared with `config` are non-execution-time and non-compile-time-constant so any use of them in a function or procedure declaration or invocation renders it non-compile-time-constant, but does not cause it to be execution-time.

Subprogram declarations or invocations may not be both compile-time-constant and execution-time since if the conditions for "execution-time" are met then the conditions for "compile-time-constant" cannot be met, and vice versa.

8.3 Evaluation of expressions

8.3.1 Execution-time expressions

$\texttt{I}_{\texttt{XYKC}}$ Expressions in ASL are either *execution-time* expressions or *non-execution-time* expressions.

$\texttt{D}_{\texttt{ZPMF}}$ An expression is an execution-time expression if either:

- it contains an execution-time storage element identifier
- it contains an execution-time function or getter invocation

8.3.2 Compile-time-constant expressions

$\texttt{I}_{\texttt{XSFY}}$ Expressions in ASL are either *compile-time-constant* expressions or *non-compile-time-constant* expressions.

$\texttt{D}_{\texttt{XRBT}}$ An expression is a compile-time-constant expression if each one of its atomic expressions is one of:

- a literal constant
- a compile-time-constant storage element identifier
- an immutable storage element identifier with a compile-time-constant initializer expression.
- compile-time-constant function or getter invocations

8.4 Behavior of types

8.4.1 Execution-time types

I$_{\text{WVGG}}$ Types in ASL are either *execution-time* types or *non-execution-time* types.

D$_{\text{JLJD}}$ A type is an execution-time type if its structure depends on either:

- an execution-time type
- an execution-time expression

Example

```
func execType(wid: integer)
begin
    // structure of R's type depends on execution time value `wid`
    var R: bits(wid);
end
```

8.4.2 Compile-time-constant types

I$_{\text{KKQY}}$ Types in ASL are either *compile-time-constant* types or *non-compile-time-constant* types.

D$_{\text{MTQJ}}$ A type is a compile-time-constant type if its structure depends only on:

- compile-time-constant types
- compile-time-constant expressions

Example

```
// All of the following are compile-time types

constant wid = 32;

type busTy of bits(wid);
type recTy of record {bus: busTy, valid: bit};

func constType()
begin
    var I: integer;
    var R: bits(wid);
end
```

I$_{\text{YBGL}}$ Note that a type may be neither an execution-time nor a compile-time-constant, For example `bits(wid)` where `wid` is a global config identifier.

8.5 Evaluation of expressions

8.5.1 Evaluation order

R_{XKGC} It is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators &&, ||, `-->`, are guaranteed to evaluate from left to right.

I_{YMRT} An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

I_{QJTN} For any function call $F(e_1, ...e_m)$, tuple $(e_1, ...e_m)$, or operation e_1 *op* e_2 (with the exception of &&, || and `-->`), it is an error if the subexpressions conflict with each other by:

- both writing to the same variable.
- one writing to a variable and the other reading from that same variable
- one writing to a variable and the other throwing an exception
- both throwing exceptions

I_{GFZT} These conditions are sufficient but not necessary to ensure that evaluation order does not affect the result of an expression, including any side-effects.

I_{QRXP} Conditional expressions and the operations &&, ||, `-->` have short-circuit evaluation described in more detail in 8.6.1 *Boolean operations*.

8.6 Evaluation of operations

Most of the operations have their usual mathematical meanings. For example, `add_int` performs the usual (unbounded) integer addition. Some exceptions to this are listed below.

In this section, we use the operation names from tables Table 7.1, Table 7.2, Table 7.3, Table 7.4 and Table 7.5 to avoid ambiguity.

8.6.1 Boolean operations

$\mathrm{R}_{\mathrm{GQNL}}$ The operator `-->` represents the boolean *implies* operator.

$\mathrm{R}_{\mathrm{LRHD}}$ The operator `<->` represents the boolean *if and only if* operator.

Conditional expressions and the operations `and_bool` `or_bool` and `implies_bool` provide the familiar short-circuit evaluation from the C programming language. That is, the first operand of `if` is always evaluated but only one of the remaining operands is evaluated; if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated; if the first operand of `or_bool` is `TRUE`, then the second operand is not evaluated; and, if the first operand of `implies_bool` is `FALSE`, then the second operand is not evaluated. We note that relying on this short-circuit evaluation can be confusing for readers and it is recommended that an `if`-statement is used to achieve the same effect.

8.6.2 Real operations

$\mathrm{R}_{\mathrm{BNCY}}$ The exponentiation operation `exp_real(x,y)` raises x to the power of y.

Note

Since the `real` type represents the real numbers, operators such as `add_real`, `mul_real`, etc. obey the usual mathematical laws such as associativity, distributivity, commutativity, etc.

8.6.3 Integer operations

$\mathrm{R}_{\mathrm{NCWM}}$ The exponentiation operation `exp_int(x,y)` raises x to the power of y.

Note

Since the `integer` type represents the mathematical integers, operators such as `add_int`, `mul_int`, etc. obey the usual mathematical laws such as associativity, distributivity, commutativity, etc.

$\mathrm{R}_{\mathrm{VGZF}}$ The shift operations are defined as follows:

$$\begin{aligned}\mathtt{shiftleft\_int}(x,n) &= \mathtt{RoundDown}(\mathtt{Real}(x) * 2.0^{n})\\ \mathtt{shiftright\_int}(x,n) &= \mathtt{RoundDown}(\mathtt{Real}(x) * 2.0^{-n})\end{aligned}$$

where the `RoundDown` library function rounds down to negative infinity.

$\mathrm{R}_{\mathrm{THSV}}$ It is an error to shift values by negative shift amounts.

$\mathrm{R}_{\mathrm{CRQJ}}$ There are division operations `div_int` and `fdiv_int` and a remainder operation `frem_int`.

$\mathrm{R}_{\mathrm{ZTJN}}$ The operation `div_int` performs exact division. The divisor (the second operand) must be a positive integer that exactly divides the first operand.

$\mathrm{R}_{\mathrm{SVMM}}$ The operation `fdiv_int` performs division rounding towards negative infinity.

R_{WWTV} It is an error to divide by zero, or by a negative integer.

R_{GHXR} The remainder operation `frem_int` is defined as the remainder of division rounding towards negative infinity. The second operand must be a positive integer.

I_{NBCT} For example, the following table shows the results of the operations:

x	y	div_int	fdiv_int	frem_int
6	3	2	2	0
-6	3	-2	-2	0
5	3	(error)	1	2
-5	3	(error)	-2	1
6	-3	(error)	(error)	(error)

Note

Since the `integer` type represents the mathematical integers, operators such as $+$, $*$, etc. obey the usual mathematical laws such as associativity, distributivity, commutativity, etc.

8.6.4 Bitvector operations

R_{BRCM} Concatenation of multiple bitvectors is done using a comma separated list surrounded with square brackets.

Example:

```
    var T: boolean = [ '1111', '0000' ] == '11110000';
```

R_{RXYN} Conversion from bitvector to `integer` using the `UInt` and `SInt` functions, and from `integer` to bitvectors using slicing, behaves as if the `integers` had 2's complement representation.

Examples:

```
func UInt{N}(x: bits(N)) => integer {0..2^N-1}
begin
    var result: integer {0..2^N-1} = 0;
    for i = 0 to N-1 do
        if x[i] == '1' then
            result = result + 2^i;
        end
    end
    return result;
end
```

```
func SInt{N}(x: bits(N)) => integer
begin
    var result = 0;
    for i = 0 to N-1 do
        if x[i] == '1' then
          result = result + 2^i;
        end
    end
    if x[N-1] == '1' then
      result = result - 2^N;
    end
    return result;
end
```

Chapter 9
Standard library

In addition to the operations, ASL provides the following standard functions and procedures.

$\mathrm{R_{DGBM}}$ All value-returning standard functions behave as compile-time-constant, as defined in 8.2.2 *Compile-time-constant subprograms*, unless otherwise stated.

$\mathrm{R_{QSVS}}$ `Unreachable()` is considered a compile-time-constant subprogram.

$\mathrm{I_{TRPS}}$ `Unreachable()` behaves like `assert FALSE` and calling it is a dynamic error.

9.1 Standard integer functions and procedures

```
// Convert a bitvector to an unsigned integer, where bit 0 is LSB.
// This is the recommended way to convert a bit vector to an integer.
func UInt{N}(x: bits(N)) => integer {0 .. 2^N-1};

// Convert a 2s complement bitvector to a signed integer.
func SInt{N}(x: bits(N)) => integer {-(2^(N-1)) .. 2^(N-1)-1};

// Absolute value of an integer.
func Abs(x: integer) => integer
begin
    return if x>=0 then x else -x;
end

// Maximum of two integers.
func Max(a: integer, b: integer) => integer
begin
    return if a>b then a else b;
end

// Minimum of two integers.
func Min(a: integer, b: integer) => integer
begin
    return if a<b then a else b;
end

// Calculate the logarithm base 2 of the input. Input must be a power of 2.
func Log2(a: integer) => integer;


// Return true if integer is even (0 modulo 2).
func IsEven(a: integer) => boolean
begin
    return (a MOD 2) == 0;
end


// Return true if integer is odd (1 modulo 2).
func IsOdd(a: integer) => boolean
begin
    return (a MOD 2) == 1;
end
```

9.2 Standard real functions and procedures

```
// Convert integer to rational value.
func Real(x: integer) => real;

// Nearest integer, rounding towards negative infinity.
func RoundDown(x: real) => integer;

// Nearest integer, rounding towards positive infinity.
func RoundUp(x: real) => integer;

// Nearest integer, rounding towards zero.
func RoundTowardsZero(x: real) => integer;

// Absolute value.
func Abs(x: real) => real
begin
    return if x>=0.0 then x else -x;
end

// Maximum of reals.
func Max(a: real, b: real) => real
begin
    return if a>b then a else b;
end

// Minimum of reals.
func Min(a: real, b: real) => real
begin
    return if a<b then a else b;
end

// Calculate the square root of x to sf binary digits.
// The second tuple element of the return value is TRUE if the result is
// inexact, else FALSE.
func SqrtRoundDown(x: real, sf: integer) => (real, boolean);
```

9.3 Standard bitvector functions and procedures

```
// Return the concatenation of 1 or more copies of a bitvector.
func Replicate{M}(x: bits(M), N: integer) => bits(M*N)
begin
    var r: bits(M*N) = Zeros(M*N);
    for i = 0 to N-1 do
        var t: bits(M*N) = [Zeros((((N-1)-i)*M), x, Zeros(i*M)];
        r = r OR t;
    end
    return r;
end

// Return a bitvector consisting entirely of N '0' bits.
func Zeros(N: integer) => bits(N)
begin
    return 0[N-1:0];
end

// Return a bitvector consisting entirely of '1' bits.
func Ones(N: integer) => bits(N)
begin
    return NOT Zeros(N);
end

// Return true if bitvector consists entirely of '0' bits.
func IsZero{N}(x: bits(N)) => boolean
begin
    return x == Zeros(N);
end

// Return true if bitvector consists entirely of '1' bits.
func IsOnes{N}(x: bits(N)) => boolean
begin
    return x == Ones(N);
end

// Zero-extend a bitvector to the same or a wider width.
func ZeroExtend{M}(x: bits(M), N: integer) => bits(N)
begin
    assert N >= M;
    return if N > M then [Zeros(N-M), x] else x;
end

// Sign-extend a bitvector (treated as 2s complement) to the same or a wider width.
func SignExtend{M}(x: bits(M), N: integer) => bits(N)
begin
    assert N >= M;
    return [Replicate(x[M-1], N-M+1), x[M-2:0]];
end

// Extend a bitvector to a specified width, treating as signed or unsigned.
// The output width might be narrower than the input, in which case the
// function is equivalent to a bit slice.
func Extend{M}(x: bits(M), N: integer, is_unsigned: boolean) => bits(N)
begin
    return if is_unsigned then ZeroExtend(x,N) else SignExtend(x,N);
end

// Return the width of a bitvector argument, without regard to its value.
func Len{N}(x: bits(N)) => integer{N}
begin
    return N;
end

// Count the number of 1 bits in a bitvector.
func BitCount{N}(x: bits(N)) => integer{0..N}
begin
```

```
    var result: integer = 0;
    for i = 0 to N-1 do
        if x[i] == '1' then
            result = result + 1;
        end
    end
    return result;
end

// Position of the lowest 1 bit in a bitvector.
// If the bitvector is entirely zero, return the width.
func LowestSetBit{N}(x: bits(N)) => integer{0..N}
begin
    for i = 0 to N-1 do
        if x[i] == '1' then
            return i;
        end
    end
    return N;
end

// Position of the highest 1 bit in a bitvector.
// If the bitvector is entirely zero, return -1.
func HighestSetBit{N}(x: bits(N)) => integer{-1..N-1}
begin
    for i = N-1 downto 0 do
        if x[i] == '1' then
            return i;
        end
    end
    return -1;
end

// Leading zero bits in a bitvector.
func CountLeadingZeroBits{N}(x: bits(N)) => integer{0..N}
begin
    return (N - 1) - HighestSetBit(x);
end

// Leading sign bits in a bitvector. Count the number of consecutive
// bits following the leading bit, that are equal to it.
func CountLeadingSignBits{N}(x: bits(N)) => integer{0..N}
begin
    return CountLeadingZeroBits(x[N-1:1] XOR x[N-2:0]);
end

// Treating input as an integer, align down to nearest multiple of 2^y.
func AlignDown{N}(x: bits(N), y: integer{1..N}) => bits(N)
begin
    return [x[N-1,N-y], Zeros(y)];
end

// Treating input as an integer, align up to nearest multiple of 2^y.
// Returns zero if the result is not representable in N bits.
func AlignUp{N}(x: bits(N), y: integer{1..N}) => bits(N)
begin
    if IsZero(x[y-1:0]) then
        return x;
    else
        return [x[N-1:y]+1, Zeros(y)];
    end
end

// The shift functions LSL, LSR, ASR and ROR accept a non-negative shift amount.
// The shift functions LSL_C, LSR_C, ASR_C and ROR_C accept a non-zero positive shift amount.

// Logical left shift
func LSL{N}(x: bits(N), shift: integer) => bits(N)
begin
```

```
    assert shift >= 0;
    if shift < N then
        let bshift = shift as integer{0..N-1};
        return [x[(N-bshift)-1:0], Zeros(bshift)];
    else
        return Zeros(N);
    end
end

// Logical left shift with carry out.
func LSL_C{N}(x: bits(N), shift: integer) => (bits(N), bit)
begin
    assert shift > 0;
    if shift <= N then
        return (LSL(x, shift), x[N-shift]);
    else
        return (Zeros(N), '0');
    end
end

// Logical right shift, shifting zeroes into higher bits.
func LSR{N}(x: bits(N), shift: integer) => bits(N)
begin
    assert shift >= 0;
    if shift < N then
        let bshift = shift as integer{0..N-1};
        return ZeroExtend(x[N-1:bshift], N);
    else
        return Zeros(N);
    end
end

// Logical right shift with carry out.
func LSR_C{N}(x: bits(N), shift: integer) => (bits(N), bit)
begin
    assert shift > 0;
    if shift <= N then
        return (LSR(x, shift), x[shift-1]);
    else
        return (Zeros(N), '0');
    end
end

// Arithmetic right shift, shifting sign bits into higher bits.
func ASR{N}(x: bits(N), shift: integer) => bits(N)
begin
    assert shift >= 0;
    let bshift = Min(shift, N-1) as integer{0..N-1};
    return SignExtend(x[N-1:bshift], N);
end

// Arithmetic right shift with carry out.
func ASR_C{N}(x: bits(N), shift: integer) => (bits(N), bit)
begin
    assert shift > 0;
    return (ASR(x, shift), x[Min(shift-1, N-1)]);
end

// Rotate right.
func ROR{N}(x: bits(N), shift: integer) => bits(N)
begin
    assert shift >= 0;
    let cshift = (shift MOD N) as integer{0..N-1};
    return [x[0+:cshift], x[N-1:cshift]];
end

// Rotate right with carry out.
func ROR_C{N}(x: bits(N), shift: integer) => (bits(N), bit)
begin
```

```
    assert shift > 0;
    let cpos = ((shift-1) MOD N) as integer{0..N-1};
    return (ROR(x, shift), x[cpos]);
end
```

9.4 Runtime exception types

```
type runtime_exception of exception;
```

9.5 Other standard functions and procedures

```
// Print one or more arguments, to an implementation defined output channel.
// This function is provided for diagnostics and does not form part of an architectural
    ↪specification.
func print(...);

// Unreachable() is used to indicate parts of a subprogram that cannot be reached,
// as a more readable alternative to `assert FALSE`.
func Unreachable();

// Convert an integer to a decimal string, prefixing with '-' if negative.
func DecStr(x: integer) => string;

// Convert an integer to a hexadecimal string, prefixing with '-' if negative.
// The exact format of the string is implementation defined.
func HexStr(x: integer) => string;

// Convert an integer to an ASCII character.
func AsciiStr(x: integer) => string
begin
    case x of
        when {0..31, 127} => // Non-printable ASCII char, implementation defined.

        when 32  => return " "; when 33  => return "!"; when 34  => return "\"";
        when 35  => return "#"; when 36  => return "$"; when 37  => return "%";
        when 38  => return "&"; when 39  => return "'"; when 40  => return "(";
        when 41  => return ")"; when 42  => return "*"; when 43  => return "+";
        when 44  => return ","; when 45  => return "-"; when 46  => return ".";
        when 47  => return "/"; when 48  => return "0"; when 49  => return "1";
        when 50  => return "2"; when 51  => return "3"; when 52  => return "4";
        when 53  => return "5"; when 54  => return "6"; when 55  => return "7";
        when 56  => return "8"; when 57  => return "9"; when 58  => return ":";
        when 59  => return ";"; when 60  => return "<"; when 61  => return "=";
        when 62  => return ">"; when 63  => return "?"; when 64  => return "@";
        when 65  => return "A"; when 66  => return "B"; when 67  => return "C";
        when 68  => return "D"; when 69  => return "E"; when 70  => return "F";
        when 71  => return "G"; when 72  => return "H"; when 73  => return "I";
        when 74  => return "J"; when 75  => return "K"; when 76  => return "L";
        when 77  => return "M"; when 78  => return "N"; when 79  => return "O";
        when 80  => return "P"; when 81  => return "Q"; when 82  => return "R";
        when 83  => return "S"; when 84  => return "T"; when 85  => return "U";
        when 86  => return "V"; when 87  => return "W"; when 88  => return "X";
        when 89  => return "Y"; when 90  => return "Z"; when 91  => return "[";
        when 92  => return "\\"; when 93 => return "]"; when 94  => return "^";
        when 95  => return "_"; when 96  => return "`"; when 97  => return "a";
        when 98  => return "b"; when 99  => return "c"; when 100 => return "d";
        when 101 => return "e"; when 102 => return "f"; when 103 => return "g";
        when 104 => return "h"; when 105 => return "i"; when 106 => return "j";
        when 107 => return "k"; when 108 => return "l"; when 109 => return "m";
        when 110 => return "n"; when 111 => return "o"; when 112 => return "p";
        when 113 => return "q"; when 114 => return "r"; when 115 => return "s";
        when 116 => return "t"; when 117 => return "u"; when 118 => return "v";
        when 119 => return "w"; when 120 => return "x"; when 121 => return "y";
        when 122 => return "z"; when 123 => return "{"; when 124 => return "|";
        when 125 => return "}"; when 126 => return "~";

        otherwise => assert FALSE;  // Integer does not map to an ASCII character

// These functions names are reserved for future use. They have no official definition.
func FileOpen(...) => ...;
func FileWrite(...) => ...;
func FileGetC(...) => ...;
func FileRead(...) => ...;
func FileClose(...) => ...;
```

Chapter 10
Runtime Environment

An ASL runtime provides run time support within a hosting environment.

Examples of a hosting environment include an interactive interpreter, an interpreter running in batch mode, a Verilog simulator or a Linux process (native executable).

R_{CTMN} The runtime should provide a mechanism to set configuration (config) variables before the entry point (default: main).

R_{BSQR} Configuration variables are read only once the entry point is called. The runtime may enforce this.

R_{JWPH} The default entry point is the `main` function.

```
func main() => integer
begin
```

R_{CHTH} When `main` returns (without throwing an exception) the runtime should pass the return value to the hosting environment.

For example: an ASL runtime for native executables may use the return value of `main` as the exit status of the process.

I_{BKLJ} By convention a return value of zero indicates success and a return value of one indicates failure.

I_{JLGK} An alternative (non-default) entry point may be specified by the user if supported by the runtime. Not all runtimes may support alternative entry points.

R_{XNSK} Uncaught exceptions cause termination of the application by the runtime. If an exception is thrown from main it is an uncaught exception. The runtime should signal an error to the hosting environment.

I_{CXPS} Output may be generated using the `print` procedure. This takes any number of arguments, of any type, and makes best efforts to print them to diagnostic output. The format of the output is not defined.

10.1 Dynamic errors

R_{MHFW}	An assertion failure arising from the `assert` keyword is a dynamic error.
I_{WXGP}	Whether or not an assertion is tested is implementation-defined: see 6.6 *Assertion statements*.
R_{VDDG}	A call to the standard library `Unreachable()` function is a dynamic error.
R_{QYKH}	It is illegal for the evaluation of a non-execution-time expression or subprogram to cause a dynamic error (e.g. an assertion failure). A tool may treat such errors as non-dynamic errors.
R_{DPZK}	The behaviour of the runtime environment when a dynamic error occurs is implementation defined.
I_{MHYS}	An implementation might immediately terminate execution in response to a dynamic error. It might also perform optimizations, or determine possible execution states, on the basis that a dynamic error cannot occur.
R_{IRNQ}	If the runtime environment raises an exception in response to a dynamic error the exception must have the supertype `runtime_exception`.
I_{BLDK}	A non-exhaustive list of dynamic specification errors can be found in 1.1.4 *Specification errors*.

Chapter 11
Changes

This chapter describes the changes in this version of the ASL specification.

11.1 Changes in version 1.0

11.1.1 Local Variables are initialized

I$_{\text{CVDB}}$ Local variables now have a defined initialization. Specifications that read local variables before the variables' first assignment may have different behaviors.

11.1.2 Slice notation

I$_{\text{MVBT}}$ The notation for extracting a slice from a bitvector uses square brackets. The use of angle brackets is no longer permitted.

The notation `[i*:n]` is introduced, to refer to a slice at an offset scaled by its width. This replaces the `Elem` construct.

Example

```
    var v: bits(128);
    var d: bits(16);
    v[n*:8] = d[7:0];    // old syntax: Elem[v,n,8] = d<7:0>
```

11.1.3 Record type

I$_{\text{PZGZ}}$ The notation for defining a record type uses the `record` keyword.

Example

```
type flags of record {valid :: boolean, data :: bit};
```

I$_{\text{FWTW}}$ The use of the `type ... is` construct for defining a record type is removed.

Example

```
    type flags is (
        B: boolean,
        Z: boolean,
        C: boolean,
        V: boolean
    )
```

I$_{\text{JDSX}}$ The syntax for initializing a record.

Example

```
    var a = MyRecord { field1 = value1, field2 = value2 };
```

11.1.4 UNPREDICTABLE

I$_{\text{QSXM}}$ `UNPREDICTABLE` is a function.

Example

```
    UNPREDICTABLE();
```

I$_{\text{QJSV}}$ The use of `UNPREDICTABLE` as a keyword is deprecated.

Example

```
    UNPREDICTABLE;
```

11.1.5 IMPLEMENTATION_DEFINED

I$_{GBMH}$ `IMPLEMENTATION_DEFINED` is a function.

Example

```
    IMPLEMENTATION_DEFINED();
```

I$_{NDWK}$ The use of `IMPLEMENTATION_DEFINED` as a keyword is deprecated.

Example

```
    IMPLEMENTATION_DEFINED;
```

11.1.6 SEE

I$_{LWMD}$ `SEE` is a function, which takes one argument that is either a string or an identifier.

Example

```
    SEE("POP r1");

    state: SomeType;
    SEE(state);
```

I$_{HBFS}$ The use of `SEE` as a keyword is deprecated.

Example

```
    SEE "POP r1";

    state: SomeType;
    SEE state;
```

11.1.7 Implicit Variables

I$_{XKWM}$ Local variables are no longer implicitly declared. All local variables must be explicitly declared.

11.1.8 Type specifiers

I$_{MJMQ}$ Type specifiers now follow identifiers rather than precede them. Type specifiers are separated from an identifier by a colon token.

```
var myvar: bits(64);

func integer somefunc(a: integer, b: integer)
begin
    return a + b;
end
```

11.1.9 Return types

I$_{ZQVM}$ Return type specifiers for functions and getters now follow rather than precede them. Return type specifiers are separated from a function's formal arguments by a `=>` token.

Example

```
func somefunc(a: integer, b: integer) => integer
begin
    return a + b;
end
```

11.1.10 var keyword

I_{PPXZ} Variable declarations must be prefixed with the var keyword.

Example

```
var myvar: bits(64);
```

11.1.11 New BNFC grammar

I_{NWMX} The following parts of this document are now auto generated from BNFC grammar:

- BNF grammar in appendix and main body
- Token rules
- List of reserved Ids
- List of delimiters

The following changes have been applied, although these do not necessarily fulfill remaining requirements for closing the related tickets.

- Removed syntax for qualid (ASL-131)
- Removed unecessary lexical grammar description
- Removed dependence of layout on NEWLINE
- Removed non-terminal `procedure_declaration` (ASL-95)
- Removed non-terminal `instruction_definition`
- Removed non-terminal `internal_definition`
- Removed `IMPLEMENTATION_DEFINED` and other hardwired exceptions (ASL-76)
- Removed various cosmetic statement syntaxes
- Removed Inout parameter syntax (ASL-52)
- New `type` syntax with `subtype` etc. (ASL-80)
- Simplified `conditional_stmt`
- Replaced `__register` with `bits` with bitfields (ASL-147)
- New `exception` type (ASL-76)
- `boolean` no longer an Enumeration (ASL-130)
- `signal` is no longer a built-in type (ASL-182)
- Replaced `__config` with `config` (ASL-42 but maybe not approved?)
- New `pattern` syntax (ASL-31)
- New `pass` statement (ASL-16)
- New `throw` syntax (ASL-76)
- New syntax for `alt` in case_stmt (ASL-34)
- New `catch_stmt` non-terminal rule (ASL-76)
- New syntax for array types (ASL-169)
- Forbid empty getter (ASL-72)
- Allow trailing commas in enums etc. (ASL-11)
- All bits of grammar specifying lists are explicitly given as BNF rules instead of using the "..", "...", etc. syntax

11.1.12 Primitive Boolean

I_{BHTH} `boolean` is now a primitive type and no longer an enumeration.

11.1.13 Case statements

I_{PQJD} Patterns and statements in case alternatives must be delimited using `=>`.

I_{KMVJ} The optional condition token for case alternatives has been changed from `&&` to `where`.

Example

```
    case a of
        when flag where a > 10 => return 10;
    end
    return a;
```

11.1.14 Field concatenation

I_TBWS Square braces are now used instead of angle brackets for multiple field selection. E.g:

```
    myreg.<n, c, v> = '000';
```

Is now:

```
    myreg.[n, c, v] = '000';
```

Additionally, this syntax can only be used on records if each field is of type bitvector.

11.1.15 IMPLIES

I_ZWHZ The binary operator `IMPLIES` has been replaced by the operator `-->`

11.1.16 IFF

I_YMFM The binary operator `IFF` has been replaced by the operator `<->`

11.1.17 Let declarations

I_YZBT Local, immutable, runtime values can now be declared with the `let` keyword.

11.1.18 Bitvector concatenation

I_RDBH Removed the colon operator for bitvector concatenation and replaced with square bracket comma delimited list.

```
    var a = c : b : d;
```

Is now written as:

```
    var a = [c, b, d];
```

11.1.19 Primitive Operations

Bitslices are no longer a primitive operation.

11.1.20 Type system changes

- Strongly typed named types with subtypes are now defined along with type checking rules for when these may be used.
- Global, immutable, runtime values can now be declared with the `let` keyword.
- Mutable objects and statically evaluable expressions are now defined.
- Shadowing rules subsumed by 4.1 *Scope of global declarations* and 6.2.1 *Scope of local declarations*.
- Width inference removed (ASL-??)
- Rules about declaring subprograms etc. (ASL-75)
- Execution-time and compile-time-constant types

11.1.21 Reserved Keywords

Added reserved keywords as grammar rule `reserved_id`.

11.1.22 Exceptions

The predefined exceptions have been replaced by *exceptions*.

Example

The old keyword `UNDEFINED` (which is no longer a keyword) does not throw an ASL exception.

This behavior can be reproduced by defining an exception called `UNDEFINED`

```
    type UNDEFINED of exception;
```

```
    throw UNDEFINED;
```

11.1.23 Qualified Identifiers

Previous versions of ASL had a "qualid" a.k.a "qualified identifier" This has been removed. (ASL-131)

Migration can be to rename functions of the form:

```
AArch32.TakeException(...)
AArch64.TakeException(...)
```

to:

```
AArch32_TakeException(...)
AArch64_TakeException(...)
```

11.1.24 Forward Declarations

Forward declarations of subprograms are no longer required in ASL.

11.1.25 Constrained types

Added rules and descriptions for constrained integers and constrained width bitvectors.

11.1.26 Return type width inference

In older ASL, bitvector-returning functions can no longer infer the result width from the calling context. This affects some standard library functions. for example:

```
var a: bits(16) = Zeros(16);         // Zeros() not permitted
var b: bits(32) = ZeroExtend(a,32);  // ZeroExtend(a) not permitted
```

11.1.27 Exclusive-OR operator

The name of the exclusive-OR operator is now `XOR`. The token `EOR` is reserved.

11.2 Examples

This section contains examples of ASL code changes from ASL v0 to ASL v1.

11.2.1 BigEndianReverse

Current ASL v0 code:

```
bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then
        return value;
    end
    return BigEndianReverse(value<half-1:0>):BigEndianReverse(value<width-1:half>);
end
```

Changed to ASL v1:

```
func BigEndianReverse {
    width: integer {8,16,32,64,128}
    } (
    value: bits(width)
    ) => bits(width)
begin
    if width == 8 then
        return value;
    end

    let half = (width DIV 2) as integer {8,16,32,64};

    // Type checker knows half ==> width DIV 2
    // The checked type conversion is required because otherwise the
    // implicit type would be integer {4,8,16,32,64}.

    return [BigEndianReverse(value[0+:half]),
            BigEndianReverse(value[half+:half])] as bits(width);

    // Note that the actuals in these invocations are both of width "half"
    // which is integer {8,16,32,64}
    // so the invocations satisfy the constraint on the signature
    // of BigEndianReverse.

    // If the type of "half" was {4,8,16,32,64} then the invocations
    // of BigEndianReverse would not have satisfied the formal
    // argument constraint of {8,16,32,64,128} on "N".

    // Note that the type checker knows that the width of the concatenation is
    // of type bits({16,24,32,40,64,72,80,96,128})
    // and also has the value (width DIV 2) + (width DIV 2)
    // which is not necessarily equal to width, hence the checked type
    // conversion is required although a smart compiler has enough
    // information to elide it.
end

func test()
begin
    // Call site example:
    // ...
    var ttd: bits(64);
    // ...
    if tt_attr.EE == '1' then
        ttd = BigEndianReverse(ttd); // returns a bits(64)
    end
end
```

11.2.2 AlignBits

Current ASL v0 code:

```
type L1STD_t is (
     bits(5) Span,
     bits(46) L2Ptr
 )

bits(N) AlignBits(bits(N) value, integer alignment)
    integer mask = (1<<alignment)-1;
    bits(N) bit_mask = mask<N-1:0>;
    return value AND (NOT bit_mask);

test()
    L1STD_t l1std = fetch_l1std();
    integer l2_align = 5 + UInt(l1std.Span);
    bits(52) l2_base = AlignBits(l1std.L2Ptr : Zeros(6), l2_align);
end
```

Changed to ASL v1:

```
type L1STD_t of record {
    Span  : bits(5),
    L2Ptr : bits(46)
    };

func AlignBits {N} (value: bits(N), alignment: integer) => bits(N)
begin
    // N is treated as the under-constrained integer
    let mask: integer  = (1<<alignment)-1;
    let bit_mask: bits(N) = mask[0+:N]; // Legal since width matches
    return value AND (NOT bit_mask); // Legal since width matches return type
end

func test()
begin
    var l1std    : L1STD_t  = fetch_l1std();
    var l2_align : integer  = 5 + UInt(l1std.Span);
    var l2_base  : bits(52) = AlignBits([l1std.L2Ptr, Zeros(6)], l2_align);
end
```

11.2.3 FPRoundBase

Current ASL v0 code:

```
bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding,
                    boolean isbfloat16, boolean fpexc)
    assert N IN {16,32,64};  // effectively N is constrained
    bits(N) result;

    // Obtain format parameters
    // - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126;  E = 8;  F = 7;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;
    end

    // ...lots of stuff...

    result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    return result;
end
```

Changed to ASL v1:

```
func FPRoundBase (
    N           : integer {16,32,64},
    op          : real,
    fpcr        : FPCRType,
    rounding    : FPRounding,
    isbfloat16 : boolean,
    fpexc       : boolean
    ) => bits(N)
begin
    // N is a parameter as well as a formal since it is used in the
    // return type

    // No assert required since N is really constrained :)
    var result: bits(N); // result is of type "bits({16,32,64})"

    // The values assigned below must be declared here or they will not
    // be in scope outside of the "if" statement's branches

    var minimum_exp: integer {-14,-126,-1022};
    var E           : integer {5,8,11};
    var F           : integer {7,10,23,52};

    // Obtain format parameters
    // - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;
        E = 5;
        F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126;
        E = 8;
        F = 7;
    elsif N == 32 then
        minimum_exp = -126;
        E = 8;
        F = 23;
    else  // N == 64
        minimum_exp = -1022;
        E = 11;
        F = 52;
    end

    // ...lots of stuff...

    // Presumably we declare sign somewhere...
    var sign: bit;

    // At this point the type checker knows:
    // E ==> E
    // F ==> F
    // so it can't compare them, but it can do arithmetic with them:

    result = [sign,
              biased_exp[E-1:0],
              int_mant[F-1:0],
              Zeros(N-(E+F+1))];

    // Note that the invocation of `Zeros` requires its argument to be a constrained integer

    // sign               is a bits(1)
    // biased_exp[E-1:0] is a bits(E)
    // int_mant[F-1:0]    is a bits(F)
    // Zeros(N-(E+F+1))  is a bits(N-(E+F+1))
    // Hence RHS         is a bits(1+E+F+(N-(E+F+1)))
    // hence RHS         is a bits(N)
    // So type-checker is happy that RHS width always matches LHS width
    // and domain comparison is necessary for demonstrating type-satisfaction

    return result;
end
```

Note

In the above, ‘E’ and ‘F’ do not need to be immutable for the invocation of ‘Zeros’ in the assignment to ‘result’. In this particular case they cancel out. The final width is “N” which is a statically evaluable expression and hence can be compared with the width of result. If it had been “E+N” then it would not be a statically evaluable expression so the width of the RHS would not be comparable with any LHS. This may need some more rules around subexpression widths to say that the final width of an expression must be reducible to a statically evaluable expression but intermediate subexpressions need not.

Chapter 12
ASL grammar

The complete grammar for ASL is as follows.

```
<comment> ::= {"//"} { <any except newline> }
            | {"/*"} { <any including newline> } {"*/"}

<boolean_lit> ::= ( {"TRUE"} | {"FALSE"} )

<int_lit> ::= digit ('_' | digit)*

<hex_lit> ::= '0' 'x' (digit | ["abcdefABCDEF"]) ('_' | digit | ["abcdefABCDEF"])*

<real_lit> ::= digit ('_' | digit)* '.' digit ('_' | digit)*

<string_lit> ::= '"' ((char - ["\"\\"]) | ('\\' ["nt\"\\"] ) )* '"'

<bitvector_lit> ::= '\'' ["01 "]* '\''

<bitmask_lit> ::= '\'' ["01x "]* '\''

<identifier> ::= ( letter | '_' ) ( letter | '_' | digit )*

identifier_list ::= identifier "," identifier_list
                  | identifier

identifier_trailing ::= identifier

identifier_trailing_list ::= identifier_trailing "," identifier_trailing_list
                           | identifier_trailing
                           | identifier_trailing ","

program ::= decl_list

annotation ::= "@" identifier "(" null_or_expr_list ")"
```

```
decl_list ::= decl decl_list
            | decl

decl ::= annotation decl
       | type_decl
       | storage_decl
       | function_decl
       | getter_decl
       | setter_decl
       | "pragma" identifier null_or_expr_list ";"

type_decl ::= "type" identifier "of" ty "subtypes" ty ";"
            | "type" identifier "of" ty ";"
            | "type" identifier "subtypes" ty with_opt ";"

field ::= identifier ":" ty

field_list ::= field "," field_list
             |
             | field

storage_decl ::= "var" identifier ":" ty ";"
               | "var" identifier ty_opt "=" expr ";"
               | "let" identifier ty_opt "=" expr ";"
               | "constant" identifier ty_opt "=" expr ";"
               | "config" identifier ty_opt "=" expr ";"

subprogram_body ::= "begin" stmt_list "end"

function_decl ::= "func" identifier parameters_opt "(" formal_list ")" return_ty_opt
    ↪subprogram_body

return_ty_opt ::= "=>" ty
                |

args_opt ::= "[" formal_list "]"
           |

getter_decl ::= "getter" identifier parameters_opt args_opt "=>" ty subprogram_body

setter_decl ::= "setter" identifier parameters_opt args_opt "=" identifier ":" ty
    ↪subprogram_body

parameters_opt ::= "{" parameter_list "}"
                 |

parameter ::= identifier ty_opt

parameter_list ::= parameter "," parameter_list
                 |
                 | parameter

formal ::= identifier ":" ty

formal_list ::= formal "," formal_list
              |
              | formal

ty ::= identifier
     | "boolean"
     | "integer" constraint_opt
     | "real"
     | "string"
     | "bit"
     | "bits" "(" bitwidth ")" bitfields_opt
     | "enumeration" "{" identifier_trailing_list "}"
     | "(" ty_list ")"
     | "array" "[" expr "]" "of" ty
     | "record" fields_opt
     | "exception" fields_opt

constraint_opt ::= constraint
                 |

bitfields_opt ::= "{" bitfield_list "}"
                |

fields_opt ::= "{" field_list "}"
             |

with_opt ::= "with" fields_opt
           |

bitwidth ::= expr
           | "-" ":" ty
           | constraint

constraint ::= "{" constraint_range_list "}"

constraint_range ::= expr
                   | expr ".." expr

constraint_range_list ::= constraint_range "," constraint_range_list
                        | constraint_range

ty_opt ::= ":" ty
         |

ty_list ::= ty "," ty_list
          |
          | ty

bitfield_spec ::= ":" ty
                | bitfields_opt

bitfield ::= "[" slice_list "]" identifier bitfield_spec

bitfield_list ::= bitfield "," bitfield_list
                |
                | bitfield

stmt ::= annotation stmt
       | decl_stmt
       | lexpr "=" expr ";"
       | identifier "(" null_or_expr_list ")" ";"
       | "return" expr_opt ";"
       | "assert" expr ";"
       | "throw" expr_opt ";"
       | "pass" ";"
       | "if" expr "then" stmt_list elsif_list else_opt "end"
       | "case" expr "of" alt_list otherwise_opt "end"
       | "for" identifier "=" expr direction expr "do" stmt_list "end"
       | "while" expr "do" stmt_list "end"
       | "repeat" stmt_list "until" expr ";"
       | "try" stmt_list "catch" catcher_list otherwise_opt "end"
       | "pragma" identifier null_or_expr_list ";"

stmt_list ::= stmt stmt_list
            | stmt

decl_stmt ::= "var" identifier ":" ty ";"
            | "var" identifier "," identifier_list ":" ty ";"
            | "var" decl_item "=" expr ";"
            | "let" decl_item "=" expr ";"
            | "constant" decl_item "=" expr ";"

decl_item ::= identifier ty_opt
```

```
            | "(" decl_item_list ")" ty_opt
            | "[" decl_item_list "]" ty_opt
            | "-" ty_opt

decl_item_list ::= decl_item "," decl_item_list
                 | decl_item

lexpr ::= "-"
        | lexpr_atom
        | "(" lexpr_list ")"

lexpr_list ::= lexpr "," lexpr_list
             | lexpr

lexpr_atom ::= identifier
             | lexpr_atom "." identifier
             | lexpr_atom "." "[" identifier_list "]"
             | "[" lexpr_atom_list "]"
             | lexpr_atom "[" null_or_slice_list "]"

lexpr_atom_list ::= lexpr_atom "," lexpr_atom_list
                  | lexpr_atom

elsif ::= "elsif" expr "then" stmt_list

elsif_list ::= elsif elsif_list
             |

else_opt ::= "else" stmt_list
           |

alt ::= "when" pattern_list where_opt "=>" stmt_list

where_opt ::= "where" expr
            |

alt_list ::= alt alt_list
           |

otherwise_opt ::= "otherwise" "=>" stmt_list
                |

pattern ::= "-"
          | expr
          | expr ".." expr
          | "<=" expr
          | ">=" expr
          | pattern_set

pattern_set ::= "{" pattern_list "}"
              | "!" "{" pattern_list "}"
              | bitmask_lit

pattern_list ::= pattern "," pattern_list
               | pattern

direction ::= "to"
            | "downto"

catcher ::= "when" identifier ":" ty "=>" stmt_list
          | "when" ty "=>" stmt_list

catcher_list ::= catcher catcher_list
               |

expr_opt ::= expr
           |

expr_list ::= expr "," expr_list
```

```
            | expr

null_or_expr ::= expr

null_or_expr_list ::= null_or_expr "," null_or_expr_list
                    |
                    | null_or_expr

expr ::= "if" cexpr "then" expr elsif_expr_list "else" expr
       | cexpr

elsif_expr ::= "elsif" expr "then" expr

elsif_expr_list ::= elsif_expr elsif_expr_list
                  |

cexpr ::= cexpr binop_boolean cexpr_cmp
        | cexpr checked_type_constraint
        | cexpr_cmp

cexpr_cmp ::= cexpr_cmp binop_comparison cexpr_add_sub
            | cexpr_add_sub

cexpr_add_sub ::= cexpr_add_sub binop_add_sub_logic cexpr_mul_div
                | cexpr_mul_div

cexpr_mul_div ::= cexpr_mul_div binop_mul_div_shift cexpr_pow
                | cexpr_pow

cexpr_pow ::= cexpr_pow binop_pow bexpr
            | bexpr

binop_boolean ::= "&&"
                | "||"
                | "-->"
                | "<->"

binop_comparison ::= "=="
                   | "!="
                   | ">"
                   | ">="
                   | "<"
                   | "<="

binop_add_sub_logic ::= "+"
                      | "-"
                      | "OR"
                      | "XOR"
                      | "AND"

binop_mul_div_shift ::= "*"
                      | "/"
                      | "DIV"
                      | "DIVRM"
                      | "MOD"
                      | "<<"
                      | ">>"

binop_pow ::= "^"

unop ::= "-"
       | "!"
       | "NOT"

binop_in ::= "IN"

bexpr ::= unop bexpr
        | expr_term
```

```
expr_term ::= expr_atom binop_in pattern_set
            | "UNKNOWN" ":" ty
            | expr_atom

expr_atom ::= identifier
            | identifier "(" null_or_expr_list ")"
            | identifier "{" field_assignment_list "}"
            | literal_expr
            | expr_atom "[" null_or_slice_list "]"
            | expr_atom "." identifier
            | expr_atom "." "[" identifier_list "]"
            | "(" pattern_list ")"
            | "[" expr_list "]"

field_assignment ::= identifier "=" expr

field_assignment_list ::= field_assignment "," field_assignment_list
                        |
                        | field_assignment

checked_type_constraint ::= "as" ty
                          | "as" constraint

slice ::= expr
        | expr ":" expr
        | expr "+:" expr
        | expr "*:" expr

slice_list ::= slice "," slice_list
             | slice
             | slice ","

null_or_slice ::= slice

null_or_slice_list ::= null_or_slice "," null_or_slice_list
                     |
                     | null_or_slice

literal_expr ::= int_lit
               | hex_lit
               | real_lit
               | bitvector_lit
               | string_lit
               | boolean_lit

<reserved_id> ::= "AND"           | "DIV"           | "DIVRM"         | "EOR"
                | "IN"            | "MOD"           | "NOT"           | "OR"
                | "SAMPLE"        | "UNKNOWN"       | "UNSTABLE"      | "XOR"
                | "_"             | "access"        | "advice"        | "after"
                | "any"           | "array"         | "as"            | "aspect"
                | "assert"        | "assume"        | "assumes"       | "before"
                | "begin"         | "bit"           | "bits"          | "boolean"
                | "call"          | "case"          | "cast"          | "catch"
                | "class"         | "config"        | "constant"      | "dict"
                | "do"            | "downto"        | "else"          | "elsif"
                | "end"           | "endcase"       | "endcatch"      | "endclass"
                | "endevent"      | "endfor"        | "endfunc"       | "endgetter"
                | "endif"         | "endmodule"     | "endnamespace"  | "endpackage"
                | "endproperty"   | "endrule"       | "endsetter"     | "endtemplate"
                | "endtry"        | "endwhile"      | "entry"         | "enumeration"
                | "event"         | "exception"     | "export"        | "expression"
                | "extends"       | "extern"        | "feature"       | "for"
                | "func"          | "get"           | "getter"        | "gives"
                | "if"            | "iff"           | "implies"       | "import"
                | "in"            | "integer"       | "intersect"     | "intrinsic"
                | "invariant"     | "is"            | "let"           | "list"
                | "map"           | "module"        | "namespace"     | "newevent"
                | "newmap"        | "of"            | "original"      | "otherwise"
                | "package"       | "parallel"      | "pass"          | "pattern"
                | "pointcut"    | "port"        | "pragma"      | "private"
                | "profile"     | "property"    | "protected"   | "public"
                | "real"        | "record"      | "repeat"      | "replace"
                | "requires"    | "rethrow"     | "return"      | "rule"
                | "set"         | "setter"      | "shared"      | "signal"
                | "statements"  | "string"      | "subtypes"    | "template"
                | "then"        | "throw"       | "to"          | "try"
                | "type"        | "typeof"      | "union"       | "until"
                | "using"       | "var"         | "watch"       | "when"
                | "where"       | "while"       | "with"        | "ztype"

<delimiter> ::= "!"    | "!="   | "&&"   | "("    | ")"    | "*"    | "*:"   | "+"    | "+:"
              | ","    | "-"    | "-->"  | "."    | ".."   | "/"    | ":"    | ";"    | "<"
              | "<->"  | "<<"   | "<="   | "="    | "=="   | "=>"   | ">"    | ">="   | ">>"
              | "@"    | "["    | "]"    | "^"    | "{"    | "||"   | "}"
```

Alphabetical index of rules

This section contains lists of Declarations, Rules and Information items, indexed alphabetically.

Declaration Index

Rules Index

Info Index