SIEVE Intermediate Representation

See Section 7 for list of Contributors

Last Updated: 2022-04-29

Identification and History

Version	Date	Notable Changes
0.1.0	2020-09-19	Initial version as proposed jointly by WizKit and FROMAGER teams on DARPA SIEVE.
0.1.1	2020-12-16	A number of bugfixes on the prior version. • Fix typographical errors and omissions. • Add checks for matching header, `num_wires`, and etc., across multiple resources. • Eliminate unused type-checking and arity-checking attributes from the attribute grammar, because IR0 has only one meaningful type. • Add `num_wires` to the instance and witness in the text and binary grammars (sections 4 and 5).
0.2.0	2021-03-19	Remove the `num_wires` and associated input size data and add a `@delete` directive.
1.0.0	2021-07-20	Add “uniformity” features: function gates, public-index for loops, and private-condition switch-case statements.

1.0.1	2022-04-29	Fix the following issues: • Fix switch-statement semantics which required a nested switch-statement where the outer case is inactive and the inner switch-statement has no active branch to cause an unexpected witness-statement invalidity. • Constrain for-loop bounds and wire-ranges such that the first must be less than or equal to the last iteration or wire. • Add a warning that the switch-statement conversion algorithm (6.3) may produce a poorly-formed relation with certain forms of switch-statement and for-loop nesting. Add a suggestion to unroll for-loops in those conditions. • Assorted typo fixes and performance improvements.

Contents

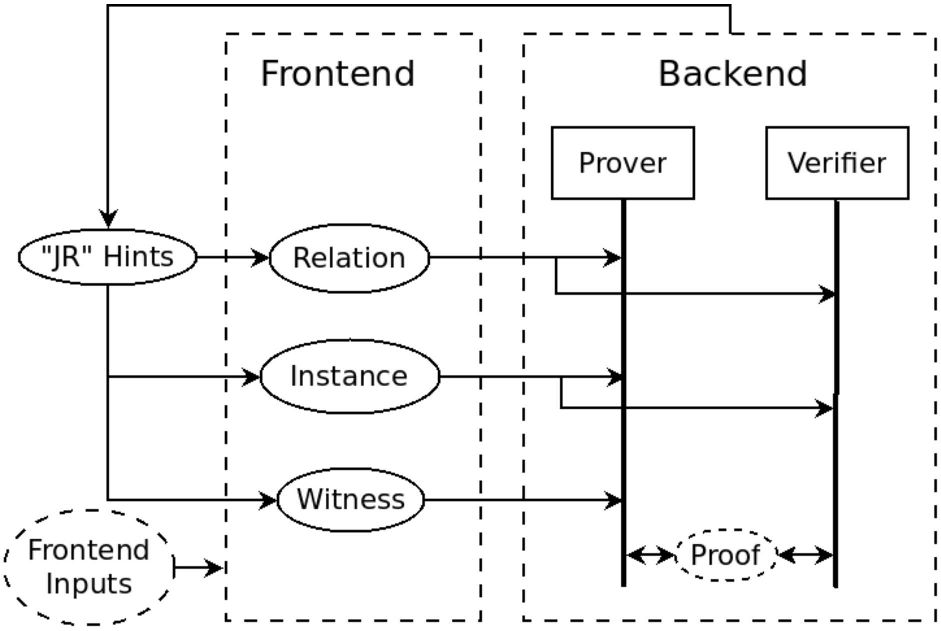
1 Introduction

This document specifies the current version of the SIEVE Intermediate Representation (IR). Changes to this specification are to be discussed and modified only through the decision process outlined in the SIEVE IR Agreement document. This document will be updated to reflect such changes.

The IR is an interaction between the front-end and the back-end portions of a Zero Knowledge (ZK) proof pipeline. The frontend transforms high level statements in a target domain into the IR. It is the producer of the three resources which are the subject of this document. The backend is the consumer of them: it is an interaction between a Prover and a Verifier, where the Prover wishes to prove a statement to the Verifier, without revealing a secret component of the proof.

At a high level, this interaction involves four (4) resources, one of which is still under development. Conceptually, each resource could be stored as a file, or transported as a stream.

- Relation - a set of directives forming a mathematical relationship between inputs. We use the language of circuitry to describe this relationship.
- Instance - common inputs to the circuit, provided by the frontend to the backend, and within the backend are available to Prover and Verifier.
- Short Witness - secret inputs to the circuit, provided by the frontend to the backend, which are later used in proof construction as secrets of the Prover. We refer to this as a "short witness" to distinguish from the Rank One Constraint System's (R1CS) witness in which each variable must have an assignment. In the SIEVE IR, only certain wires are assigned from the short witness, the others are computed by the circuit.
- "JR" Hints ("I+1 Representation") - hints from the backend to the frontend describing optimal circuit generation parameters. This resource is currently under development.



In its simplest form, the relation is a list of gate directives, each with an output wire and two input wires. The gates and their connections through wires form a circuit. This simple form is useful for streaming, where the proof system may consume a gate, process it, and immediately discard everything except the output wire. These simple circuits can process wire values in two fields, either $GF(p)$ ("arithmetic") or $GF(2)$ ("boolean"). Beyond this simple form, the IR adds features for expressing uniformity in the circuit. These features may be toggled on or off for a given relation, either enabling greater expressiveness, or guaranteeing the simple stream processing. The first of these features is "function-gates" which encapsulate functionalities to be reused. "For-loops" repeat a sub-circuit for a fixed number of repetitions. Lastly "switch-case statements" enable private conditions to be expressed directly in the IR. These constructs both compact a relation and allow proof systems to quickly recognize structures and in some cases amortize computational effort.

1.1 Concrete IR Formats

This specification defines two forms, to resolve conflicting usage patterns between human and computer use cases. To avoid ambiguity during translation from binary to text or vice-versa, special care must be taken to ensure that the text and binary formats remain isomorphic. Specifically, every element defined by one format has an equivalent in the other format.

- Text format - For human use during education, demonstration, and development.
- Binary format - For automated use in a performance critical environment.

1.2 IR Validity and Semantics

In addition to isomorphic structure, both formats should have the same semantics. This specification considers a few levels of validity, ranging from syntax and recognition through the acceptance of a proof given in the IR. Each of the following validities is considered successive over the prior.

1. **Syntactic Validity:** If a resource matches the syntactic constraints of either the Text (section 4) or Binary (section 5) formats, then it is syntactically valid.

2. **Resource Validity:** Each resource may be *well-formed* if it, in isolation, satisfies constraints given in section 4.
   - **Relation Well-formedness:** Satisfied mainly by its gates being topologically ordered, and in single static assignment (SSA) form.
   - **Instance Well-formedness** and **Witness Well-formedness:** In isolation, the validity of these resources is indicated by each value being a member of the field.

3. **Evaluation Validity:** Considers all three resources together by evaluating each directive of the relation, consuming each value from the instance and witness streams, and checking that each `$assert_zero` directive carries the value 0. If this is the case, then statement has a valid set of witness values that evaluate the circuit. In other words, when passed through a ZK backend system, the proof generated would be verified.

Resource and Evaluation Validities may be collectively referred to as *semantics*.

1.3 IR Tooling (Informational only)

We will briefly describe a few software suites for working with the SIEVE IR. They are overviewed for informational purposes only, and their inclusion is not to be construed as an endorsement of either.

WizToolKit is a toolkit and a library designed for working with the SIEVE IR in C++. It includes parsing APIs for both the text and binary formats along with tools for checking *resource validity* and *evaluation validity* (`wtk-firealarm`), converting between the text and binary formats (`wtk-press`), and visualizing relations in graph form (`wtk-viz`). *WizToolKit* can be found in the WizToolKit Repository.

zkInterface-SIEVE is an implementation of the FlatBuffers-based SIEVE IR and a set of tools used to enable interoperability between teams. The library extends the open-source library, which integrates several R1CS frontends and backends. It includes API for producing and consuming relations in the IR binary format, and allows for parsing and printing JSON and YAML textual formats. The library includes a `validator` tool for checking *resource validity*, an `evaluator` tool for checking *evalutation validity*, a `metrics` tool for checking different stats about the circuit, as well as converting to and from the existing *zkInterface* R1CS relations. *zkInterface-SIEVE* can be found in the zkInterface-sieve repository.

2 Example

A few samples of the SIEVE IR are given in this section.

2.1 Point on Curve

The following table shows an example arithmetic circuit. The relation, instance, and witness are shown in their own columns. The header resource is shown as an element of each of the other resources.

Relation	**Instance**	**Witness**
This relation will check that a given point is a point on a Montgomery curve. $By^2 = x^3 + Ax^2 + x$	A, and B, defining the curve are given as the instance – available to both Prover and Verifier.	x, and y, the point on the curve are the witness – available only to the Prover.

// Header Start
version 1.0.0;
field
characteristic 97
degree 1;
// Header End
relation
gate_set: arithmetic;
features: simple;
@begin
$2 <- @short_witness; // x
$3 <- @mul($2, $2); // x^2
$4 <- @mul($2, $3); // x^3
$0 <- @instance; // A
$5 <- @mul($3, $0); // Ax^2
$6 <- @add($4, $5);
$7 <- @add($6, $2); // RHS
@delete($0);
@delete($2, $6);
$8 <- @short_witness; // y
$9 <- @mul($8, $8); // y^2
$1 <- @instance; // B
$10 <- @mul($9, $1); // LHS
@delete($8, $9);
$11 <- @mulc($7,<96>);
$12 <- @add($11,$10);
@assert_zero($12);
@end
	// Header Start
	version 1.0.0;
	field
	characteristic 97
	degree 1;
	// Header End
	instance @begin
	< 15 > ;
	< 4 > ;
	@end
		// Header Start
		version 1.0.0;
		field
		characteristic 97
		degree 1;
		// Header End
		short_witness @begin
		< 2 > ;
		< 39 > ;
		@end

2.2 Fibonacci

These excerpts of IR code calculate the first ten (10) Fibonacci numbers. Notably, the number of repetitions is *not* private, it is a constant of the circuit. Two versions are presented, one using an anonymous function as the body of the for-loop, another using a named function.

Anonymous For-Loop Body

```
version 1.0.0;
field characteristic 97 degree 1;
relation
gate_set: arithmetic;
features: @for;
@begin
$0 <- <1>;
$1 <- <1>;
$2...$10 <- @for i @first 2 @last 10
  $i <- @anon_call($(i - 1), $(i - 2),
      @instance: 0, @short_witness: 0)
    $0 <- @add($1, $2);
  @end
@end
@end
```

Named For-Loop Body

```
version 1.0.0;
field characteristic 97 degree 1;
relation
gate_set: arithmetic;
features: @function, @for;
@begin
@function(add_2, @out: 1, @in: 2,
    @instance: 0, @short_witness: 0)
  $0 <- @add($1, $2);
@end
$0 <- <1>;
$1 <- <1>;
$2...$10 <- @for i @first 2 @last 10
  $i <- @call(add_2, $(i - 1), $(i - 2));
@end
@end
```

2.3 Matrix Multiplication

This example demonstrates the multiplication of two matrices. It reads matrix M from the instance and matrix N from the witness. It also reads C from the instance. Then it computes C' as $M * N$ and asserts that $C = C'$.

```
version 1.0.0;
field characteristic 97 degree 1;
relation
```

```
gate_set:arithmetic;
features:@for,@function;
@begin
@function(sum, @out: 1, @in: 4, @instance: 0, @short_witness: 0)
  // out: $0
  // in: $1 ... $4
  $5 <- @add($1, $2);
  $6 ... $6 <- @for i @first 2 @last 2
    $(i + 4) <- @anon_call($(1 + i), $(3 + i),
        @instance: 0, @short_witness: 0)
      $0 <- @add($1, $2);
    @end
  @end
  $0 <- @add($4, $6);
@end

// M (3x4): $0...$11 (instance)
// N (4x5): $27...$46 (witness)
// C (3x5): $12...$26 (instance)
// C' (3x5): $47...$61

// Computes the product C':=M*N, and checks that C'==C

$0 ... $11 <- @for i @first 0 @last 11
  $i <- @anon_call(@instance: 1, @short_witness: 0)
    $0 <- @instance;
  @end
@end
$12 ... $26 <- @for i @first 12 @last 26
  $i <- @anon_call(@instance: 1, @short_witness: 0)
    $0 <- @instance;
  @end
@end
$27 ... $46 <- @for i @first 27 @last 46
  $i <- @anon_call(@instance: 0, @short_witness: 1)
    $0 <- @short_witness;
  @end
@end

$47 ... $61 <- @for i @first  0 @last 2
  $(47 + (i * 5)) ... $(51 + (i * 5)) <- @anon_call(
      $0 ... $11, $27 ... $46,
      @instance: 0, @short_witness: 0)
    // C'[i][...]: $0 ... $4
    // M: $5 ... $16
    // N: $17 ... $36
    $0 ... $4 <- @for j @first 0 @last 4
      $j <- @anon_call($5 ... $16, $17... $36,
          @instance: 0, @short_witness: 0)
        // C'[i][j]: $0
        // M: $1 ... $12
        // N: $13 ... $32
        $33 ... $36 <- @for k @first 0 @last 3
          $(k + 33) <- @anon_call(
              $(1 + ((i * 4) + k)), $(13 + ((k * 5) + j)),
              @instance: 0,@short_witness: 0)
            $0 <- @mul($1, $2);
          @end
        @end
        $0 <- @call(sum, $33 ... $36);
      @end
    @end
  @end
@end

// Check that C' == C
@for i @first 0 @last 2
  @anon_call(
      $((i * 5) + 12) ...$((i * 5) + 16),
      $((i * 5) + 47) ... $((i * 5) + 51),
      @instance: 0, @short_witness: 0)
    // C[i][...]: $0 ... $4
    // C'[i][...]: $5 ... $9
    @for j @first 0 @last 4
      @anon_call($j, $(j + 5), @instance: 0, @short_witness: 0)
        $2 <- @mulc($0, < 96 >);
        $3 <- @add($1, $2);
        @assert_zero($3);
      @end
    @end
  @end
@end
@end
```

3 IR Overview and Abstract Syntax

This section will describe the SIEVE IR at an overview level and should be sufficient to grant a workable understanding of the IR. Section 4 will provide authoritative textual syntax and IR semantics; Section 5 will provide binary syntax. Mentioned previously, the IR has three (3) "resources" which may be thought of as files but could also be provided via other methods – e.g. streams, shared memory, and so on. These resources take the R(x; w) form: R is the relation, x is the instance, and w is the witness.

The IR's relation represents a circuit using sequence of directives to manage a sparsely populated lists of wire values. A value is described by its index in the list, and most directives will insert a new value at an output index. A wire's index is represented as an integer in the range of 0 and $2^{64} - 1$. Although a directive may reference or insert a value at any index, using sequential indexes can improve performance. The lists are considered sparse for two reasons. First, wires may be removed from the list to conserve memory. Second, the minimal constraints on output index usage means that gaps may form in the wires list, although this is discouraged.

The instance and witness resources are streams of values. Certain directives in the relation will consume a value from either stream.

This overview uses "template" texts, surrounded with boxes, and where "template expressions" may be repeated and replaced by concrete expressions described elsewhere. In general, whitespace is optional and ignored, except for a single whitespace element delimiting non-obvious token sequences. Template expressions are italicized with surrounding delimiters with dark green text, for example {*expression*}. All template expressions will be described as appropriate literal values, or sub expressions. Repetition of a template expression is indicated with an ellipsis (...), also in green. Tables following each template will describe the semantics of each template.

3.1 Resource Header Overview

The first component of the IR is a header block with evaluation parameters for the circuit. The header is shared by all resources. There are two parameters defined by the header: a version number for quick recognition of the IR, and the computation field.

```
version {major}.{minor}.{patch};

field
  characteristic {p}
  degree 1;
```

Header Semantics

Template Expression	Name	Description
$\{major\}$ $\{minor\}$ $\{patch\}$	Version	The version of the IR is given in semantic version form. This must match one of the versions given in the Identification and History section of this document. Syntactically, these parameters must be positive unbounded integers, in decimal form, with no leading zeros.
$\{p\}$	Field	The field governs the domain in which proofs may be expressed. It may be either $GF(p)$ or $GF(2)$ (a special case of the former). The syntax gives a characteristic (p), and a degree (n) which would describe a field $GF(p^n)$. However the degree is fixed to one (1). The p variable may be an unbounded integer, and must be prime.

3.2 Instance and Short Witness Overview

Syntactically each of the instance and short witness is a sequence of field elements, acting as a stream. Certain directives in the relation will consume a value from one of these streams. If values in either stream are exhausted this is a failure of *evaluation validity*. If values remain in a stream after processing then this is also an *evaluation invalidity*.

```
{header}

{type} @begin
  < {c} > ; ...
@end
```

Instance and Witness Semantics

Template Expression	Description	Syntactic Constraints	Semantic Constraints
$\{header\}$	A header, as described in section 3.1, is required here.	See section 3.1.	The $\{p\}$ variable is captured in this section.
$\{type\}$	Indicates whether this resource is the instance or witness.	Must be the exact text `instance` or `short_witness`.	

{c}...	Each {c} is a literal constant specified as a gate input.	A numeric literal, which is syntactically positive and unbounded.	Must be in the range of 0 up to $\{p\}$.

3.3 Relation Overview: Structure, Gate Set, and Feature Toggles

The relation starts by describing its own contents, first by which gates it may use (the *gate set*) then by which uniformity features it may use (*feature toggles*). Following these is a list of function gate declarations, a list of the named function gates allowed in the relation, along with function gate definitions. Lastly is the body of the relation, the list of directives making up the circuit.

```
{header}
relation
gate_set: {gateset};
features: {features};
@begin
  {functions}...
  {directive}...
@end
```

Each template expression is described in this table.

Template Expression	Name	Description
{header}	Header	A header, as described in section 3.1, is required here.
{gateset}	*Gate set*	The *gate set* describes the allowable gate directives of the circuit. The actual gate directives will be described in section 3.5. There are two *canonical gate sets*, either `arithmetic` or `boolean`. If a canonical gate set is not desired, then a *partial gate set* may described as the enumeration of gate names. A *partial gate set* is defined by a comma separated subset of the gates from a *canonical gate set*. For `arithmetic`, these would be `@add`, `@addc`, `@mul`, and `@mulc`; and for `boolean` gates these would be `@and`, `@not`, and `@xor`. If an *partial gate set* enumeration were empty, it would be considered a semantic error.

{*features*}	*Feature Toggles*	The feature toggles are a list of optional features which may be enabled or disabled by their presence. If a feature is present, it must appear in a comma separated set. The allowed features are `@function`, `@for`, and `@switch` The toggles for `@for` and `@switch` each describes whether for loops (subsection 3.6) or switch-case statements (subsection 3.7) are enabled. The toggle for `@function` indicates whether function gates (subsection 3.5) are allowed, except for anonymous functions within the body of a for loop or switch-case statement. If no features are enabled, then the keyword `simple` should take the feature list's place. Unless the *feature toggle* is `simple`, wires between 2^{63} and $2^{64} - 1$ are reserved as "ephemeral wires", meaning that the frontend may not emit them. Instead, the backend may insert them as needed.
{*functions*}...	Function Gate Declarations	This is a list of function gate declarations. Each declares a named function gate which may be used as a {*directive*}. Further description will be given in subsection 3.5
{*directive*}...	Directives	This is a list of directives, each directive will be described in further subsections 3.4, 3.5, 3.6, and 3.7.

3.4 Relation Overview: Simple Gate Directives

Most of the relation is described through lists of directives. "Gate directives" perform calculations on their inputs and produce an output. In general, gate directives will look like the assignment of the calculation's result to a wire. There are three allowed forms for a gate directive.

- **Binary** directives have two wire inputs, and one output. They are written as

```
${x} <- {calculation} ( ${y} , ${z} );
```

- **Binary constant** directives have one wire input, one constant input, and one output. They are written as

```
${x} <- {calculation} ( ${y} , < {c} > );
```

- **Unary** directives have one wire input, and one output. They are written as

```
${x} <- {calculation} ( ${y} );
```

The allowed *{calculation}*s for each directive vary by the profile given in the header. Each of the given function names can be considered a keyword in the IR language.

- Under the `boolean` *canonical gate set*, the following calculations are defined.
  - `@and` is a binary directive which computes logical and.
  - `@xor` is a binary directive which computes logical exclusive or.
  - `@not` is a unary directive which computes logical negation.
- Under the `arithmetic` *canonical gate set*, the following calculations are defined.
  - `@mul` is a binary directive which computes field multiplication.
  - `@add` is a binary directive which computes field addition.
  - `@mulc` is a binary constant directive which computes field multiplication.
  - `@addc` is a binary constant directive which computes field addition.

In both profiles, five special directives, called the *common directives*, are allowed: Copy, Assignment, Assert Zero, Input (instance and short_witness), and Deletion, as indicated below. Regardless of *gate set*, these directives are always allowed.

- **Copy** will duplicate the value of one variable into another variable.

```
  ${x} <- ${y};
```

- **Assignment** will give one variable a constant value.

```
  ${x} <- < {c} >;
```

- **Assert Zero** will check that a variable is equivalent to zero (0), otherwise it constitutes an *evaluation invalidity*.

```
  @assert_zero( ${y} );
```

- **Input** directives have a single output wire, and read the next value of the named stream. The *{input-stream}* may be either of `@short_witness` or `@instance`, indicating which of the input streams to read from. These may be considered keywords of the IR language.

```
  ${x} <- {input-stream};
```

- **Deletion** will remove a wire or a range of wires from the wires list.

```
  @delete( ${y} );
  @delete( ${a}, ${b} );
```

The following table describes each of the template expressions used in this subsection.

Template Expression	Description	Syntactic Constraints	Semantic Constraints
$\{x\}$	A single directive output wire specified as a 0-based index.	A numeric literal in the range of $[0; 2^{64})$.	No $\{x\}$ wire may appear more than once, even if the $\{x\}$ has been deleted previously. Unless the *feature toggle* is `simple`, $\{x\}$ must not exceed 2^{63}
$\{y\}$ $\{z\}$	A single directive input wire specified as a 0-based index.	A numeric literal in the range of $[0; 2^{64})$.	A wire $\{y\}$ or $\{z\}$ must have been assigned a value before appearing as a $\{y\}$ or a $\{z\}$ and must not have been previously deleted. Unless the *feature toggle* is `simple`, $\{y\}$ or $\{z\}$ must not exceed 2^{63}
$\{c\}$	A literal constant specified as a gate input.	A numeric literal which is syntactically an unbounded positive integer.	$\{c\}$ must be in the range of 0 up to but not including $\{p\}$.
$\{a\}$, $\{b\}$	A range of wires starting at $\{a\}$ and concluding at $\{b\}$, both inclusive.	A numeric literal in the range of $[0; 2^{64})$.	Each wire between $\{a\}$ and $\{b\}$ must have previously been assigned a value and must not have been previously deleted. Unless the *feature toggle* is `simple`, no wire in this range must exceed 2^{63}.

3.5 Relation Overview: Function Gates

The function gate feature adds a function declaration and two directives to the IR.

Declaration Declares a named function gate for later use. These must be listed together at the beginning of the relation.

Invocation Invokes a previously declared function gate.

Anonymous Invocation Define a function gate at the point of its invocation.

Should a backend wish not to support function gates directly, they may be "inlined", replacing their invocations with a copy of their body. A procedure for this is outlined in section 6.1.

Function Gate Declaration

Function gates are declared ahead of time and invoked later. Their declarations indicate the number of wires entering and exiting the function-gate, as well as the number of instance and short witness values it will consume.

```
@function({name}, @out: {out}, @in: {in},
    @instance: {n_instance}, @short_witness: {n_s_witness})
  {directives}...
@end
```

Function Gate Declaration Semantics

Template Expression	Description	Syntactic Constraints	Semantic Constraints
$\{name\}$	The name by which this function gate will be referenced.	This is a C-like identifier, but may also use `.` or `::` for grouping similar functions.	The name must not have previously been used in another function gate declaration.
$\{out\}$	The number of outputs wires from the subcircuit.	Integer literal between 0 and $2^{64} - 1$.	Gate directives may output on wires 0 through $\{out\} - 1$ to assign the subcircuit's output wires.
$\{in\}$	The number of input wires from the subcircuit.	Integer literal between 0 and $2^{64} - 1$.	Gate directives may input from wires $\{out\}$ through $\{in\} + \{out\} - 1$ to consume the subcircuit's input wires.

{n_instance}	The number of instance values consumed the subcircuit.	Integer literal between 0 and $2^{64}-1$.	{directives}... must have enough @instance directives or function gate invocations so that the @instance directive count plus all the invocations' {n_instance}s is {n_instance}, exactly.
{n_s_witness}	The number of short witness values consumed by the subcircuit.	Integer literal between 0 and $2^{64}-1$.	{directives}... must have enough @short_witness directives or function gate invocations so that the @short_witness directive count plus all the invocations' {n_s_witness}s is {n_s_witness}, exactly.
{directives}...	A list of directives (other than more function gate definitions) forming the body of the function gate.	These are the same directives from IR0 as allowed by the *gate set*. Additionally, IR1 directives may be used, except for more function gate declarations.	Gate directives may assign to wires {in} + {out} or higher to create wires local to the subcircuit.

Function Gate Invocation

A function gate may be invoked as follows. An invocation may occur within the top-level circuit, or within a subcircuit. Note however, if a function-gate is invoked from within a function-gate, it must have been declared lexically ahead of the current one, specifically preventing direct or indirect recursion.

```
{out-list} <- @call({name}, {in-list});
```

If a function gate has no inputs or no outputs, then the invocation may take one of the following forms. An invocation could also have no inputs and no output wires, presumably if it consumed from @short_witness or @instance for input, and used @assert_zero in lieu of output wires.

```
@call({name}, {in-list});

{out-list} <- @call({name});

@call({name});
```

Function Gate Invocation Semantics

Template Expression	Description	Syntactic Constraints	Semantic Constraints
{name}	The name of the function to be invoked.	This is a C-like identifier, but may also use `.` or `::` for grouping similar functions.	The name must have previously been used as a function gate declaration.
{out-list}	A list of wires which will connect the output wires of the subcircuit.	Must be a wire list, as described below.	The number of wires in the list must be equal to the {out} parameter of the function gate declaration. Each wire in the list must be distinct. The wires in this list must not have previously been assigned. These wires will be assigned after the invocation.

{*in-list*}	A list of wires which will connect the output wires of the subcircuit.	Must be a wire list, as described below.	The number of wires in the list must be equal to the {*in*} parameter of the function gate declaration. Wires in this list may be duplicates, although this is not recommended. The wires in this list must have previously been assigned.

The wire list syntax used for {*out-list*} and {*in-list*} are lists of either single indexes or index ranges. Multiple indexes and index ranges may be separated by commas.

Single ${*x*}, where {*x*} is an integer literal. The wire list will contain the wire {*x*}.

Range ${*x*}...${*y*}, where {*x*} and {*y*} are integer literals. The wire list will contain every wire between {*x*} and {*y*} (inclusive).

Function Gate Invocation Example

This example demonstrates a simple function gate computing the sum of four inputs.

```
version 1.0.0;
field characteristic 97 degree 1;
relation
gate_set: arithmetic;
features: @function;
@begin
  @function(add_4, @out: 1, @in: 4,
      @instance: 0, @short_witness: 0)
    $5 <- @add($1, $2);
    $6 <- @add($3, $4);
    $0 <- @add($5, $6);
  @end

  $0 <- @short_witness;
  $1 <- @instance;
  $2 <- @short_witness;
  $3 <- @instance;
  $4 <- @call(add_4, $0...$3);
  @assert_zero($4);
@end
```

Anonymous Invocations

An anonymous invocation declares an unnamed function gate, and invokes it in the same directive. It has the following form.

```
{out-list} <- @anon_call({in-list},
    @instance: {n_instance}, @short_witness: {n_s_witness})
  {directives}...
@end
```

Just like a named function invocation, an anonymous function may have no inputs or no outputs.

```
@anon_call({in-list}, @instance: {n_instance},
    @short_witness: {n_s_witness})
  {directives}...
@end

{out-list} <- @anon_call(@instance: {n_instance},
    @short_witness: {n_s_witness})
  {directives}...
@end

@anon_call(@instance: {n_instance}, @short_witness: {n_s_witness})
  {directives}...
@end
```

The following template expressions are related to those of the Function Gate Declaration (3.5) or Function Gate Invocation (3.5).

Template Expression	Explanation
{name}	The {name} is not used by an anonymous invocation.
{out}, {in}	These template expressions of a Declaration are replaced with the length of the {out-list} and {in-list} wire lists.
{n_instance}, {n_s_witness}	These have the same usage in a Declaration and an Anonymous Invocation.
{directives}...	This has the same usage in a Declaration and an Anonymous Invocation.
{out-list} {in-list}	These have the same usage in an Invocation and an Anonymous Invocation.

Anonymous Invocation Example

This example demonstrates an anonymous function computing the sum of four inputs.

```
version 1.0.0;
field characteristic 97 degree 1;
relation
gate_set: arithmetic;
features: @function;
@begin
  $0 <- @short_witness;
  $1 <- @instance;
  $2 <- @short_witness;
  $3 <- @instance;
  $4 <- @anon_call($0...$3, @instance: 0, @short_witness: 0)
    $5 <- @add($1, $2);
    $6 <- @add($3, $4);
    $0 <- @add($5, $6);
  @end
  @assert_zero($4);
@end
```

3.6 Relation Overview: For Loops

A for loop repeats the invocation of a function gate. The invocation may be anonymous, but this is not required. It has the following form.

Should a backend wish not to support for loops directly a procedure for “unrolling” a loop is described in section 6.2.

```
{assign-list} <- @for {iterator} @first {first} @last {last}
  {invocation}
@end
```

A for loop may omit the {*assign-list*} so long as its {*invocation*} has no output wires. In this case the {*invocation*} must also not have an {*out-list*}.

```
@for {iterator} @first {first} @last {last}
  {invocation}
@end
```

For Loop Semantics

Template Expression	**Description**	**Syntactic Constraints**	**Semantic Constraints**
{*assign-list*}	A list of wires, in the scope containing the for loop, which are assigned by the for loop.	The same wire list form as used by Function Gate Invocation (3.5).	Before the loop, all wires in the list must be unassigned. After the loop all of them will have been assigned.
{*iterator*}, {*first*}, {*last*}	These control the number of iterations of the loop. {*iterator*} is a loop iterator (see below), while {*first*} and {*last*} are start and stop conditions.	{*iterator*} is an identifier (see below). {*first*} and {*last*} may be iterator expressions (see below).	{*iterator*} increments by one (1) on each iteration of the loop. On the first iteration, {*iterator*} carries the value {*first*}, and on the last it carries the value {*last*}.
{*invocation*} (and transitively {*out-list*} and {*in-list*})	This is a function gate invocation, anonymous or not.	The wire indexes in the invocation's {*out-list*} and {*in-list*} may be replaced with iterator expressions (see below).	The union of each iteration's {*out-list*} must form an equivalent set to the {*assign-list*}. Each iteration's {*out-list*} must be distinct from that of each other iteration.

Iterator Expressions

Iterator expressions enable the input and output wires of a function gate invocation to differ on each iteration of the loop. Each iterator expression is a simple arithmetic expressions using the loop {*iterator*}s of any lexically containing loop. The expressions replace wire indexes for the {*invocation*}'s inputs and outputs. An iterator expression can take one of the following forms, and represents the index of a wire, using integers between 0 and $2^{64}-1$. Loop {*iterator*}s have lexical scope. Functionally, this means that on scope boundaries entering an anonymous function they are preserved, but entering a named function they are not. For further clarity refer to section 4.7.

- A numeric constant.
- A "loop iterator", which refers to the value of a loops iteration counter. Syntactically this is a C-style identifier, but may also use `.` or `::` for grouping similar functions. See Appendix A for details.
- An expression of "sub-expressions". This takes the form `(`{*a*} {*op*} {*b*}`)`. Where {*a*} and {*b*} are "sub- expressions", and {*op*} is one of the following operations. The operations have wraparound behavior from $2^{64}-1$ to 0.

  - +: integer addition.
  - -: integer subtraction.
  - *: integer multiplication.

- A division constant expression. This takes the form *{a}* / *{b}* where *{a}* is a "sub-expression" and *{b}* is a numeric constant. The result is the 64-bit unsigned integer quotient.

Examples

Here are a few example "iterator-expressions", as well an example of Fibonacci. Note that for Fibonacci, the "n" is a public constant (the bounds of the loop).

- $123 is the exact wire.
- $i or $(i + 123) indexes an "array" within a for-loop.
- $((i * 10) + j) indexes a "square array" within a pair of for-loops.

```
$2 ... $10 <- @for i @first 2 @last 10
  $i <- @anon_call($(i - 1), $(i - 2),
      @instance: 0, @short_witness: 0)
    $0 <- @add($1, $2);
  @end
@end
```

To clarify lexical *{iterator}* scoping consider the following two example nested loop excerpts. The first nests using anonymous functions for the outer loop body. The second uses a named function, and because some of the inner loop's iterator expressions use the outer loop's iterator, an error is caused.

```
 /* ... */ <- @for i @first 0 @last 10
   /* ... */ <- @anon_call(/* ... */)
     @for j @first 0 @last 10
       $((i * 10) + j) <- @anon_call(/* ... */)
         /* ... */
       @end
   @end
 @end
```

The equivalent using named functions is however erroneous.

```
@function(foo, /* ... */)
  /* ... */ <- @for j @first 0 @last 10
    $((i * 10) + j) <- @anon_call(/* ... */) // Error
```

```
      /* ... */
    @end
  @end
@end

/* ... */ <- @for i @first 0 @last 10
  /* ... */ <- @call(foo, /* ... */)
@end
```

3.7 Relation Overview: Switch-Case Statements

The switch-case directive grants the IR the capability to evaluate branches, as is ubiquitous in conventional computing. However, in order to protect the the short witness, performance characteristics cannot always be mirrored. If the length or structure of one branch differs from other branches, then the verifier could use such information to learn about the witness. Thus, every branch is processed, and only the result of the matching branch is chosen as the "selected" result.

Should a backend wish not to support a switch-case directly, a procedure for conversion to a selection circuit, or "multiplexing", is described in section 6.3.

```
{assign-list} <- @switch ( ${condition} )
  @case < {case} > : {invocation}
  @case < {case} > : {invocation}
  @case < {case} > : {invocation}
  ...
@end
```

Switch-Case Semantics

The *{condition}* is the wire index of an assigned wire, whose value is matched against each *{case}*. Where the *{condition}*'s value matches a *{case}*, the corresponding *{invocation}*'s assignments are kept as the *{assign-list}*'s assignments.

If any two *{case}*s have the same value, then the relation is semantically invalid. If the *{condition}* does not match any *{case}*, the witnessed statement is invalid.

Template Expression	Description	Syntactic Constraints	Semantic Constraints
{assign-list}	A list of wires, in the scope containing the switch-case, which are assigned by the switch-case.	The same wire list form as used by Function Gate Invocation (3.5).	Before the switch-case, all wires in the list must be unassigned. Afterwards all of them will have been assigned.

{*condition*}, {*case*}	These select the "selected" case of the switch-case statement.	The {*condition*} is a wire-number, prefixed by `$`, and in the range of 0 through $2^{63} - 1$. Each {*case*} is a field element, delimited with `<` and `>`.	The {*condition*} must have been previously assigned, and each {*case*} must be in the range of 0 through $p-1$. No {*case*} may be duplicated within a same switch-case. If the {*condition*}'s value is not matched by a case, this is an *evaluation invalidity*.
{*invocation*} (and transitively {*out-list*}, {*in-list*}, and {*directives*})	Each case is backed by a function gate invocation, either named or anonymous.	The {*out-list*} and `<-` are omitted from each invocation, otherwise, these are syntactically identical to a named or anonymous function-gate invocation.	The {*assign-list*} must have the same length as the {*out-list*} would have. The "selected" case's {*out-list*} is replaced with the {*assign-list*}, while each other case's {*out-list*} is replaced with wires which can be considered discarded. A {*directive*} which has side-effects must have special meaning described in subsection 3.7.

Side-Effect Directives in a Switch-Case Statement

As certain directives have effects other than assignment of wires, their semantics within a switch-case must be modified. Specifically `@assert_zero` and the input directives, `@instance` and `@short_witness`, have altered semantics.

The `@assert_zero` directive causes an *evaluation invalidity* if its given wire does not carry the value zero (0). This effect is undesirable in non-selected cases, thus the effect is disabled in the body of each case, except the selected case.

The input directives (`@instance` and `@short_witness`) cause streams to be advanced on each occurrence, and each branch might use a different number of inputs. If the IR were to match exactly the expectation of conventional branching, then differing stream consumption counts could reveal the switch condition, breaking zero-knowledge. However, if the IR were to trace each branch and consume every input on each branch, this could be wasteful and preclude certain interesting optimizations. Instead, the totality of the switch-case must consume the maximum number of inputs regardless of which case. This could be implemented by duplicating the stream before each case (demonstrated by the reference semantics in subsection 4.8), rewinding between cases, or replacing input directives with function-gate inputs (outlined in subsection 6.3). Note that due to "maximum consumption", regardless of which case is selected, the overall stream length does not vary with case selections.

Using the "maximum consumption" may seem to produce nonsensical values in non-selected cases. This is allowable, because the nonsensical values are ignored as both function gate outputs and as `@assert_zero` inputs.

It is worth mentioning that that there may be up to $p-1$ unique cases within a switch. In some cases it may be undesirable, impractical, or impossible to enumerate each case of the switch-case. This leaves open the possibility for a witnessed-statement to have a switch-case statement with out a selected branch. This possibility is defined as a witnessed-statement invalidity, meaning that backends must be careful to avoid malicious behavior.

Example

This function implements a vectorized/simd style private arithmetic operation, with trivial encoding: $0 \to mul$, $1 \to add$. Here the logic is fully unrolled for practicality due to the brevity of the implicit loop. Input vectors are stored in wires $1...$4 and $5..$8.

```
$9 ... $12 <- @switch ($0)
    @case <0>:
        @anon_call($1 ... $8, @instance:0, @short_witness:0)
        $0 <- @mul($4, $8)
        $1 <- @mul($5, $9)
        $2 <- @mul($6, $10)
        $3 <- @mul($7, $11)
        @end
    @case <1>:
        @anon_call($1 ... $8, @instance:0, @short_witness:0)
        $0 <- @add($4, $8)
        $1 <- @add($5, $9)
        $2 <- @add($6, $10)
        $3 <- @add($7, $11)
        @end
@end
```

3.8 Supporting additional Syntax

In the following sections 4 and 5, we concretely specify the two syntax serializations so that proposed features or tradeoffs can be more easily expressed in a rigorous manner.

4 Textual Serialization and Authoritative Semantics

The textual serialization and exact semantics of IR0 are described here using Extended BNF Grammar, Donald Knuth's Attribute Grammar concept, and clarifying comments and section headers. `Green text` notates syntax in EBNF, and `purple text` notates semantics, using pseudo code to assign and evaluate grammatical attributes. In some cases, where pseudo code both produces *inherited* attributes and requires *synthesized* attributes of a sub-element, a `process(⟨element`$_n$`⟩);` directive indicates when the sub-element produces its *synthesized* attributes.

The `purple` semantic sections are split into two subsections, one for checking *resource validity* of an individual resource, the other for checking *evaluation validity* or evaluating the circuit. The *resource validity* subsections are surrounded by `resource { ... }` blocks, and the *evaluation validity* subsections by `evaluation { ... }` blocks. In each, a `fail_if(condition)` pseudo function indicates a condition which could render the syntax element invalid. The `resource` subsection should be considered a prerequisite of the `evaluation` subsection, thus to fully evaluate a circuit, one would concatenate the two subsections.

Throughout this section, ⟨`wire-number`⟩, ⟨`field-literal`⟩, and ⟨`label`⟩ elements will be referenced. They are defined in Appendix A. Syntactically, a ⟨`wire-number`⟩ is a number prefixed with a `$`, for example $123. The ⟨`wire-number`⟩ must be in the range of 0 through $2^{64} - 1$, although with certain features enabled, wires 2^{63} through $2^{64} - 1$ are reserved as "ephemeral wires" for used by a backend as needed. Semantically, a ⟨`wire-number`⟩ names a wire and is the index into the *`wires`* list. A ⟨`field-literal`⟩ is the literal form of a field element. These are notated by a number surrounded by angle brackets, for example < 456 >. The number must be a member of $GF(p)$ defined by prime p given in the header. a ⟨`label`⟩ is a C-style identifier used to label elements of the program state other than wires. Concrete definitions for ⟨`wire-number`⟩, ⟨`field-literal`⟩, and ⟨`label`⟩ elements are given in Appendix A: Textual Syntax Building Blocks, along with syntax for comments which are ignored.

4.1 Resource Header

The header is the first element of the IR. All resources share the same header. It declares just a version number and the computation field.

```
⟨header⟩ ::=
            ⟨version-decl⟩
            ⟨field-decl⟩
```

The version number is given using the Major.Minor.Patch form from semantic versioning. It must match one of the versions listed in the IR's Identification and History. Embedding a version number into the IR enables tooling to quickly warn the user when unsupported features may be present. Semantically, the version is largely unimportant other than in for checking consistency across resources.

```
⟨version-decl$_0$⟩ ::= ‘version’ ⟨decimal-literal$_0$⟩
                  ‘.’ ⟨decimal-literal$_1$⟩
                  ‘.’ ⟨decimal-literal$_2$⟩ ‘;’
resource {
  Version.major := ⟨decimal-literal$_1$⟩.value;
  Version.minor := ⟨decimal-literal$_2$⟩.value;
  Version.patch := ⟨decimal-literal$_3$⟩.value;
}
```

The field declaration introduces a global variable which is used to validate field elements. The *p* variable is a prime modulus, or the field's characteristic. Although the field degree is required, it is fixed at 1.

```
⟨field-decl$_0$⟩ ::= ‘field’
                 ‘characteristic’ ⟨numeric-literal$_1$⟩
                 ‘degree’ ⟨numeric-literal$_2$⟩ ‘;’
resource {
  p := ⟨numeric-literal$_1$⟩.value;
  fail_if(⟨numeric-literal$_2$⟩.value != 1);
  fail_if(!is_prime(p));
}
```

4.2 Instance and Short Witness

The instance and witness resources provide streams of field elements which are consumed by directives of the relation. Each is similarly structured as a simple list of field elements. Abstractly, each stream is modeled as a “first-in-first-out” queue data structure. The *instance* and *shortWitness* queues/streams will be inherited attributes of the relation's directives.

```
⟨instance$_0$⟩ ::= ⟨header⟩
               ‘instance’ ‘@begin’
                 [ ⟨field-literal$_{1...n}$⟩ ‘;’ ]*
               ‘@end’
resource {
  for i from 1 to n {
    fail_if(p <= ⟨field-literal$_i$⟩.value);
  }
}
evaluation {
  for i from 1 to n {
    instance.put(⟨field-literal$_i$⟩.value);
  }
}
```

```
⟨short-witness_0⟩ ::= ⟨header⟩
                    ‘short_witness’ ‘@begin’
                      [ ⟨field-literal_{1...n}⟩ ‘;’ ]*
                    ‘@end’
resource {
  for i from 1 to n {
    fail_if(p <= ⟨field-literal_i⟩.value);
  }
}
evaluation {
  for i from 1 to n {
    shortWitness.put(⟨field-literal_i⟩.value);
  }
}
```

4.3 Circuit State

During *resource* and *evaluation validity* the circuit carries a state, and each directive modifies the state. Some elements of the state are used in both *resource* and *evaluation validity* phases, some only during one of the phases. The primary element of the program state is the *wires* set described in subsection 4.3.1 The *wires* set is used in both *resource* and *evaluation validity* phases. In addition, an *iterators* map tracks the for-loop iterators (see subsection 4.7) which are lexically in scope, along with their current values. These are bundled into a *state* object will be passed through directives as an inherited attribute.

```
struct state {
  wireset wires;
  map iterators;
```

During the *evaluation validity* phase, the instance and short witness input streams are added to the state. These are not available during *resource validation*, because this phase occurs independently for each resource. The *evaluation validity* phase also requires a flag indicating if the current circuit element is part of a switch statement's inactive case, it is normally `true`, and `false` when within the inactive case.

```
  stream instance;
  stream shortWitness;
  bool switchActive;
}
```

4.3.1 Wires Set

The the *wires* set keeps the circuit's active wires in a sparse list. Generally speaking, wires may be inserted, retrieved, and removed in any order, with the following restrictions.

- Once a wire is inserted, it may not be replaced, even after it has been removed (single static assignment).
- A wire may be retrieved or removed only after it has been inserted (topological ordering).
- After being removed, a wire may not be retrieved or removed again.

Scoping mechanics allow a "containing scope" to map wires into a "sub scope". When a new scope is opened, a new *wires* set object is created and wires are mapped from the containing scope into the sub scope as either outputs or inputs. In addition to these input and output wires, a scope may have local wires.

Input Previously assigned wires of a containing scope which may be used as inputs to a directive.

Output Unassigned wires of a containing scope which may be first assigned as outputs of a directive. After the scope closes, the containing scope may use these wires as inputs to a directive, as may the sub scope after the initial assignment.

Local wires assigned within a scope which are not its output. Once the scope closes, these wires become inaccessible.

Here we will define an abstract datatype for the *wires* set. A single wire is represented as the field element value which it carries along with some flags to indicate its lifetime.

```
struct wire {
  bool assigned;
  bool deleted;
  field_element value;
}
```

A `wireset` encapsulates the scoping rules previously described. It is a mapping from indexes to wires. All index arithmetic is performed using unsigned 64-bit integer logic with wraparound from $2^{64} - 1$ to 0 and reverse. There are two contiguous lists output wires and then input wires mapped from higher scopes as referenced objects. Wires with higher indexes than the inputs and outputs are local wires. Their storage and retrieval are modeled with a map data type to allow for discontinuities. Implementations are encouraged to come up with alternate representations, so long as the `wireset`'s behavior is equivalent.

```
class wireset {
  list<wire> outputs;
  list<wire> inputs;
  map<uint64 : wire> locals;

  wire findIndex(uint64 index) {
    if(index < outputs.size) {
      return outputs[index];
    }
    else if(index < outputs.size + inputs.size) {
      return inputs[index - outputs.size];
    }
    else if(locals.has(index - (outputs.size + inputs.size))) {
      return locals.get(index - (outputs.size + inputs.size));
    }
    else {
      return null;
    }
  }
```

The following operations for insertion and retrieval which may seem atomic are split due to distinct `resource` and `evaluation` validity phases. Remember, the *resource validity* phase should be considered a prerequisite for the *evaluation validity* phase.

```
  void retrieveResource(uint64 index) {
    wire w = findIndex(index);
    fail_if(w == null);
    fail_if(w.deleted || !w.assigned);
  }

  field_element retrieveEvaluation(uint64 index) {
    wire w = findIndex(index);
    return w.value;
  }

  void insertResource(uint64 index) {
    if(index < outputs.size) {
      fail_if(outputs[index] == null);
      fail_if(outputs[index].assigned);
      outputs[index].assigned = true;
    }
    else {
      fail_if(index < outputs.size + inputs.size);
      fail_if(locals.has(index - (outputs.size + inputs.size)));

      locals.put(index - (outputs.size + inputs.size),
        wire(assigned := true, deleted := false, value := null);
    }
  }

  void insertEvaluation(uint64 index, field_element value) {
    wire w = findIndex(index);
    w.value = value;
  }
```

The remove operation is only performed during the *resource validity* phase. Removal capabilities are also limited only to local wires. Note, to make explicit that deleted wires may not have their indexes reused, they are simply marked as unused, indicating that they may not be reused. This is done only for clarity of specification; implementations are encouraged to come up with alternate methods of enforcement.

```
  void remove(uint64 index) {
    fail_if(index < output.size + input.size);
    fail_if(!locals.has(index - (output.size + input.size));
    wire w = locals.get(index - (output.size + input.size));
    fail_if(w.deleted || !w.assigned);
    w.deleted = true;
  }
```

The following methods are used when remapping wires from a containing scope to sub scope, and take action during the *resource validity* phase. Consider them to be methods of the containing scope, with the sub scope as parameters. These are operations of the *resource validity* phase. Mapping additional wires into a sub scope after local wires have been assigned may throw off indexing, and is thus considered erroneous; no syntax to do so is provided, so this may also be a redundant check.

```
  void remapInput(index, wireset subScope) {
    fail_if(subScope.locals.size != 0);

    wire w = findIndex(index);
    fail_if(w == null);
    fail_if(w.deleted || !w.assigned);

    subScope.inputs.append(w);
  }

  void remapOutput(index, wireset subScope) {
    fail_if(subScope.locals.size != 0);

    insertResource(index);

    wire w = findIndex(index);
    w.assigned = false;

    subScope.outputs.append(w);
  }
```

To process all cases of a switch-case statement, and choose the correct result, we use this method for mapping "dummy" outputs wires into a sub-scope (during the *resource validity* phase) and later copying a "dummy" wire onto an actual output wire (shown in section 4.8).

```
  void mapDummies(count) {
    fail_if(outputs.size != 0);

    for i from 0 to count {
      outputs.append(
        wire(assigned := false, deleted := false, value := null));
    }
  }
}
```

Within later subsections, the `__Resource` methods will assure that single static assignment and topological ordering are respected before the `__Evaluation` methods perform an action.

4.3.2 Global State

In addition to the *state* attribute, there are a few global variables which carry program state. Notably *FunctionsMap* retrieves a function definition, given its name. The p prime also defines the field of computation. Additional global variables are be declared for feature toggles, etc.

Most global variables are limited to a single resource. However, the global variables *Version* and p are declared in the header which is duplicated across resources. Before the `evaluation` phase, their equivalence across resources must be checked for them to have *evaluation validity*.

```
  evaluation {
    fail_if(relation.Version != instance.Version
    || relation.Version != short_witness.Version);
    fail_if(relation.p != instance.p
      || relation.p != short_witness.p);
  }
```

4.4 Relation: Structure, Gate Set, and Feature Toggles

Relation Structure

The relation lists directives, collectively forming a circuit. The relation begins with a ⟨header⟩, the *gate set* the *feature toggles*, function-gate definitions (if enabled), and finally the circuit body. function-gate definitions will be described in section 4.6, and the circuit's body is a list of directives, described later in this section. Shown here also is setup for the *state* attribute.

```
⟨relation⟩ ::= ⟨header⟩ ‘relation’
               ⟨gate-set⟩ ⟨feature-toggles⟩
               ‘@begin’
                 [ ⟨function-declare⟩ ]*
                 ⟨directive-list_0⟩
               ‘@end’
resource {
  ⟨directive-list_0⟩.state.wires = wireset();
  ⟨directive-list_0⟩.state.iterators = map();
}
evaluation {
  ⟨directive-list_0⟩.state.instance =
    /* The instance resource */;
  ⟨directive-list_0⟩.state.shortWitness =
    /* The short witness resource */;
  ⟨directive-list_0⟩.state.switchActive = true;
}
```

Gate Set

The *gate set* controls which gates/computations are available in a circuit. Canonically, a gate set may be one of `boolean` (consisting of `@and`, `@xor`, and `@not` gates) or `arithmetic` (consisting of `@mul`, `@mulc`, `@add`, and `@addc` gates). The *gate set* defines global flags which guard usage of gate-directives in later

```
⟨gate-set⟩ ::= ‘gate_set’ ‘:’ [ ‘boolean’ | ‘arithmetic’ ] ‘;’
resource {
  match {
    case ‘boolean’ : {
      EnableAdd  := false;
      EnableAddC := false;
      EnableMul  := false;
      EnableMulC := false;
```

```
        EnableAnd := true;
        EnableNot := true;
        EnableXor := true;
      }
      case ‘arithmetic’ : {
        EnableAdd := true;
        EnableAddC := true;
        EnableMul := true;
        EnableMulC := true;

        EnableAnd := false;
        EnableNot := false;
        EnableXor := false;
      }
    }
  }
```

Additionally *partial gate sets* are allowed to forbid certain gates from one of the *canonical gate sets*. To create a *partial gate set*, the *gate set's* value is replaced with a list of all remaining allowed gates. The ability to form *partial gate sets* does not imply that a circuit must use every gate in its *gate set*.

```
⟨arithmetic-gate-name⟩ ::= ‘@add’ | ‘@addc’ | ‘@mul’ | ‘@mulc’
⟨gate-set⟩ ::= ‘gate_set’ ‘:’
             ⟨arithmetic-gate-name_1⟩ [ ‘,’ ⟨arithmetic-gate-name_2...n⟩ ]* ‘;’
resource {
  EnableAdd := false;
  EnableAddC := false;
  EnableMul := false;
  EnableMulC := false;

  EnableAnd := false;
  EnableNot := false;
  EnableXor := false;

  for i from 1 to n {
    match(⟨arithmetic-gate-name_i⟩) {
      ‘@and’ : { EnableAnd := true; }
      ‘@not’ : { EnableNot := true; }
      ‘@xor’ : { EnableXor := true; }
    }
  }
}
```

```
⟨boolean-gate-name⟩ ::= '@and' | '@not' | '@xor'
⟨gate-set⟩ ::= 'gate_set' ':'
             ⟨boolean-gate-name_1⟩ [ ',' ⟨boolean-gate-name_2...n⟩ ]* ';'
resource {
  EnableAdd  := false;
  EnableAddC  := false;
  EnableMul  := false;
  EnableMulC  := false;

  EnableAnd  := false;
  EnableNot  := false;
  EnableXor  := false;

  for i from 1 to n {
    match(⟨boolean-gate-name_i⟩) {
      '@add'  : { EnableAdd  := true; }
      '@addc' : { EnableAddC  := true; }
      '@mul'  : { EnableMul  := true; }
      '@mulc' : { EnableMulC  := true; }
    }
  }
}
```

Feature Toggles

Feature toggles control the usage of features beyond simple gate directives. Syntactically the toggles form a list of enabled features, if none are desired simple should be used instead. Each toggle creates a global variable indicating if the feature may be used. In later subsections, usage of these features will be guarded by global flags defined here. Note that for loops and switch-case statements depend on function gates. In the case that either for loops or switch-case statements are enabled but function gates are not, then a single anonymous function will be allowed as the body of each for loop or case block.

In the case that a backend does not wish to support one of the features directly, procedures for "flattening" each to a equivalent circuit, without the feature, are given in section 6.

```
⟨feature-name⟩ ::= '@function' | '@for' | '@switch'
⟨feature-toggles⟩ ::= 'features' ':'
                    ⟨feature-name_1⟩ [ ',' ⟨feature-name_2...n⟩ ]*  ';'
resource {
  FunctionToggle  := false;
  ForLoopToggle  := false;
  SwitchCaseToggle  := false;
  for i from 1 to n {
    match(⟨feature-names$_i$⟩) {
      ‘@function’: { FunctionToggle := true; }
      ‘@for’:      { ForLoopToggle := true; }
      ‘@switch’:   { SwitchCaseToggle := true; }
    }
  }
}
```

```
⟨feature-toggles⟩ ::= ‘features’ ‘:’ ‘simple’ ‘;’
resource {
  FunctionToggle := false;
  ForLoopToggle := false;
  SwitchCaseToggle := false;
}
```

Circuit Body

The ⟨directive-list⟩ lists directives in both the circuit body as well as the bodies of sub-scope features. Each directive is given the inherited *state* attribute as described in subsection 4.3. Directives are not necessarily evaluated in order. They may be reordered so long as doing so does not alter the meaning of the circuit or trigger a `fail_if(...)` which wouldn't otherwise be triggered. For example, switching two `@short_witness` directives, changes the order in which an input stream is consumed, and thus would be forbidden.

In order to check that a function-gate consumes all the instance and short witness values which it claims, the ⟨directive-list⟩ must synthesize a count for each.

```
⟨directive-list$_0$⟩ ::= [ ⟨directive$_{1...n}$⟩ ]+
resource {
  for i from 1 to n {
    ⟨directive$_i$⟩.state := ⟨directive-list$_0$⟩.state;
  }
  process(⟨directive$_{1...n}$⟩);

  ⟨directive-list$_0$⟩.instanceCount := 0;
  ⟨directive-list$_0$⟩.shortWitness := 0;
  for i from 1 to n {
    ⟨directive-list$_0$⟩.instanceCount += ⟨directive$_i$⟩.instanceCount;
    ⟨directive-list$_0$⟩.shortWitnessCount += ⟨directive$_i$⟩.shortWitnessCount;
  }
}
```

Each allowable directive is also given the inherited *state* attribute. These directives will be described in detail in subsections 4.5, 4.6, 4.7, and 4.8.

```
⟨directive$_0$⟩ ::= ⟨binary-gate$_1$⟩
              | ⟨binary-const-gate$_1$⟩
              | ⟨unary-gate$_1$⟩
              | ⟨input$_1$⟩
              | ⟨copy$_1$⟩
              | ⟨assign$_1$⟩
              | ⟨assert-zero$_1$⟩
              | ⟨delete-single$_1$⟩
              | ⟨delete-range$_1$⟩
              | ⟨function-invoke$_1$⟩
              | ⟨anon-function$_1$⟩
              | ⟨for-loop$_1$⟩
              | ⟨switch-statement$_1$⟩
resource {
  match {
    ⟨binary-gate$_1$⟩: {
      ⟨binary-gate$_1$⟩.state := ⟨directive$_0$⟩.state;
      process(⟨binary-gate$_1$⟩);
    }
    ...
    ⟨switch-statement$_1$⟩: {
      ⟨switch-statement$_1$⟩.state := ⟨directive$_0$⟩.state;
      process(⟨switch-statement$_1$⟩);
    }
  }

  ⟨directive$_0$⟩.instanceCount := 0;
  ⟨directive$_0$⟩.shortWitnessCount := 0;
  match {
    ⟨input$_1$⟩: {
      ⟨directive$_0$⟩.instanceCount += ⟨input$_1$⟩.instanceCount;
      ⟨directive$_0$⟩.shortWitnessCount += ⟨input$_1$⟩.shortWitnessCount;
    }
    ⟨function-invoke$_1$⟩: {
      ⟨directive$_0$⟩.instanceCount += ⟨function-invoke$_1$⟩.instanceCount;
      ⟨directive$_0$⟩.shortWitnessCount +=
        ⟨function-invoke$_1$⟩.shortWitnessCount;
    }
    ⟨anon-function$_1$⟩: {
      ⟨directive$_0$⟩.instanceCount += ⟨anon-function$_1$⟩.instanceCount;
      ⟨directive$_0$⟩.shortWitnessCount += ⟨anon-function$_1$⟩.shortWitnessCount;
    }
    ⟨for-loop$_1$⟩: {
      ⟨directive$_0$⟩.instanceCount += ⟨for-loop$_1$⟩.instanceCount;
      ⟨directive$_0$⟩.shortWitnessCount += ⟨for-loop$_1$⟩.shortWitnessCount;
    }
    ⟨switch-statement$_1$⟩: {
      ⟨directive$_0$⟩.instanceCount += ⟨switch-statement$_1$⟩.instanceCount;
      ⟨directive$_0$⟩.shortWitnessCount +=
        ⟨switch-statement$_1$⟩.shortWitnessCount;
    }
  }
}
```

4.5 Relation: Simple Gate Directives

The first directive is a ⟨binary-gate⟩ which performs some function with two input wires, and one output wire. ⟨binary-gate-type⟩ will be defined below, but briefly may be @and or @xor in a boolean circuit, or @add or @mul in an arithmetic circuit.

```
⟨binary-gate$_0$⟩ ::= ⟨wire-number$_1$⟩ ‘<-’ ⟨binary-gate-type$_1$⟩ ‘(’
                ⟨wire-number$_2$⟩ ‘,’ ⟨wire-number$_3$⟩ ‘)’ ‘;’
resource {
  ⟨binary-gate$_0$⟩.state.wires.retrieveResource(⟨wire-number$_2$⟩.index);
  ⟨binary-gate$_0$⟩.state.wires.retrieveResource(⟨wire-number$_3$⟩.index);
  ⟨binary-gate$_0$⟩.state.wires.insertResource(⟨wire-number$_1$⟩.index);
}
evaluation {
  ⟨binary-gate$_0$⟩.state.wires.insertEvaluation(⟨wire-number$_1$⟩.index,
      ⟨binary-gate-type$_1$⟩.computation(
        ⟨binary-gate$_0$⟩.state.wires.retrieveEvaluation(
          ⟨wire-number$_2$⟩.index)
        ⟨binary-gate$_0$⟩.state.wires.retrieveEvaluation(
          ⟨wire-number$_3$⟩.index)));
}
```

A ⟨binary-const-gate⟩ performs some function with one input wire, one constant input, and one output wire. ⟨binary-const-gate-type⟩ will be defined below, but briefly may be @addc or @mulc in an arithmetic circuit. Note the ⟨field-literal⟩ element will self-check that it is in range of $[0, p)$.

```
⟨binary-const-gate$_0$⟩ ::= ⟨wire-number$_1$⟩ ‘<-’ ⟨binary-const-gate-type$_1$⟩ ‘(’
                      ⟨wire-number$_2$⟩ ‘,’ ⟨field-literal$_1$⟩ ‘)’ ‘;’
resource {
  ⟨binary-const-gate$_0$⟩.state.wires.retrieveResource(⟨wire-number$_2$⟩.index);
  fail_if($p$ <= ⟨field-literal$_1$⟩.value);
  ⟨binary-const-gate$_0$⟩.state.wires.insertResource(⟨wire-number$_1$⟩.index);
}
```

```
evaluation {
  ⟨binary-const-gate_0⟩.state.wires.insertEvaluation(⟨wire-number_1⟩.index,
      ⟨binary-const-gate-type_1⟩.computation(
        ⟨binary-const-gate_0⟩.state.wires
          .retrieveEvaluation(⟨wire-number_2⟩.index)
        ⟨field-literal_1⟩.value));
}
```

A ⟨unary-gate⟩ performs some function with one input wire, and one output wire. The function which it performs, ⟨unary-gate-type⟩, will be defined below, but briefly must only be `@not` for a boolean circuit.

```
⟨unary-gate_0⟩ ::= ⟨wire-number_1⟩ ‘<-’ ⟨unary-gate-type_1⟩ ‘(’
                ⟨wire-number_2⟩ ‘)’ ‘;’
resource {
  ⟨unary-gate_0⟩.state.wires.retrieveResource(⟨wire-number_2⟩.index);
  ⟨unary-gate_0⟩.state.wires.insertResource(⟨wire-number_1⟩.index);
}
evaluation {
  ⟨unary-gate_0⟩.state.wires.insertEvaluation(⟨wire-number_1⟩.index,
      ⟨unary-gate-type_1⟩.computation(
        ⟨unary-gate_0⟩.state.wires.retrieveEvaluation(
          ⟨wire-number_2⟩.index)));
}
```

Syntax and semantic definitions for ⟨binary-gate-type⟩, ⟨binary-const-gate-type⟩, and ⟨unary-gate-type⟩ are provided here.

```
⟨binary-gate-type_0⟩ ::= ‘@and’ | ‘@xor’ | ‘@add’ | ‘@mul’
resource {
  fail_if(‘@and’ && !EnableAnd);
  fail_if(‘@xor’ && !EnableXor);
  fail_if(‘@add’ && !EnableAdd);
  fail_if(‘@mul’ && !EnableMul);
}
evaluation {
  ⟨binary-gate-type_0⟩.computation := match {
    case ‘@and’: /* logical and */;
    case ‘@xor’: /* logical exclusive or */;
    case ‘@add’: /* addition in GF(p) */;
    case ‘@mul’: /* multiplication in GF(p) */;
  }
}
```

```
⟨binary-const-gate-type$_0$⟩ ::= ‘@addc’ | ‘@mulc’
resource {
  fail_if(‘@addc’ && !EnableAddC);
  fail_if(‘@mulc’ && !EnableMulC);
}
evaluation {
  ⟨binary-const-gate-type$_0$⟩.computation := match {
    case ‘@addc’: /* addition in GF(p) */;
    case ‘@mulc’: /* multiplication in GF(p) */;
  }
}
```

```
⟨unary-gate-type$_0$⟩ ::= ‘@not’
resource {
  fail_if(!EnableNot);
}
evaluation {
  ⟨unary-gate-type$_0$⟩.computation := /* logical negate */;
}
```

The ⟨input⟩ directive assigns a value to its output wire by popping a value from the aforementioned *instance* or *shortWitness* queue.

```
⟨input$_0$⟩ ::= ⟨wire-number$_1$⟩ ‘<-’ [ ‘@instance’ | ‘@short_witness’ ] ‘;’
resource {
  ⟨input$_0$⟩.state.wires.insertResource(⟨wire-number$_1$⟩.index);
  match {
    ‘@instance’: {
      ⟨input$_0$⟩.instanceCount := 1;
      ⟨input$_0$⟩.shortWitnessCount := 0;
    }
    ‘@short_witness’: {
      ⟨input$_0$⟩.instanceCount := 0;
      ⟨input$_0$⟩.shortWitnessCount := 1;
    }
  }
}
evaluation {
  if(‘@instance’) {
    fail_if(⟨input$_0$⟩.state.instance.size == 0);
    ⟨input$_0$⟩.state.wires.insertEvaluation(
      ⟨wire-number$_1$⟩.index,
      ⟨input$_0$⟩.state.Instance.pop());
  }
  else {
    fail_if(⟨input_0⟩.state.shortWitness.size == 0);
    ⟨input_0⟩.state.wires.insertEvaluation(
      ⟨wire-number_1⟩.index,
      ⟨input_0⟩.state.ShortWitness.pop());
  }
}
```

The ⟨copy⟩ directive will copy the value from its input wire to the output wire, without changing it.

```
  ⟨copy_0⟩ ::= ⟨wire-number_1⟩ ‘<-’ ⟨wire-number_2⟩ ‘;’
  resource {
    ⟨copy_0⟩.state.wires.retrieveResource(⟨wire-number_2⟩.index);
    ⟨copy_0⟩.state.wires.insertResource(⟨wire-number_1⟩.index);
  }
  evaluation {
    ⟨copy_0⟩.state.wires.insertEvaluation(⟨wire-number_1⟩.index,
        ⟨copy_0⟩.state.wires.retrieveEvaluation(⟨wire-number_2⟩.index));
  }
```

The ⟨assign⟩ directive will assign the value from an input constant to the output wire. The ⟨field-literal⟩ element self checks that it is in range of $[0, p)$.

```
  ⟨assign_0⟩ ::= ⟨wire-number_1⟩ ‘<-’ ⟨field-literal_1⟩ ‘;’
  resource {
    ⟨assign_0⟩.wires.insertResource(⟨wire-number_1⟩.index);
  }
  evaluation {
    ⟨assign_0⟩.wires.insertEvaluation(
      ⟨wire-number_1⟩.index, ⟨field-literal_1⟩.value);
  }
```

To verify an output value, the ⟨assert-zero⟩ directive checks that the value of some wire is exactly zero (0). Failing this check would constitute an *evaluation invalidity*.

```
  ⟨assert-zero_0⟩ ::= ‘@assert_zero’ ‘(’ ⟨wire-number_1⟩ ‘)’ ‘;’
  resource {
    ⟨assert-zero_0⟩.state.wires.retrieveResource(⟨wire-number_1⟩.index);
  }
  evaluation {
    if(⟨assert-zero_0⟩.state.switchActive) {
```

```
    fail_if(⟨assert-zero_0⟩.state.wires
      .retrieveEvaluation(⟨wire-number_1⟩.index) != 0);
  }
}
```

The ⟨delete-single⟩ and ⟨delete-range⟩ directives are hints to a backend that certain wires may be removed from active memory. Although this hint may be ignored by a backend, it does constitute a promise that these wires will never again be referenced or reassigned. Note that the wires given in ⟨delete-range⟩ include both the first wire, the last wire, and each wire between.

```
⟨delete-single_0⟩ ::= ‘@delete’ ‘(’ ⟨wire-number_1⟩ ‘)’ ‘;’
resource {
  ⟨delete-single_0⟩.state.wires.remove(⟨wire-number_1⟩.index);
}

⟨delete-range_0⟩ ::= ‘@delete’ ‘(’ ⟨wire-number_1⟩ ‘,’ ⟨wire-number_2⟩ ‘)’ ‘;’
resource {
  fail_if(⟨wire-number_1⟩.index > ⟨wire-number_2⟩.index);
  for i from ⟨wire-number_1⟩.index to ⟨wire-number_2⟩.index {
    ⟨delete-range_0⟩.state.wires.remove(i);
  }
}
```

4.6 Relation: Function Gates

A function-gate separates a sub-circuits declaration from its invocation or invocations. The function-gate declarations are listed in the top level scope before the circuit's body (see section 4.4). Invocations take the form of directives referencing the name of a prior declaration. Additionally, an "anonymous function" may be declared and invoked simultaneously. Since the "anonymous function" has no name, it cannot be referenced in multiple locations.

Function gates represent sub-circuits, not functions in the conventional sense. As such, function gates cannot recurse. A special processing phase is added to their attribute grammars to detect recursion. This phase should be interpreted as processing a resource from front to back, without reordering. Although described here as a prerequisite to *resource validity*, it is indeed a component to *resource validity*. It relies on a *FunctionsSet* global set to check previously encountered functions. Additionally the *resource* and *evaluation validity* phases of a function gate delayed from their declaration. Instead of processing attributes immediately, they are cached in a global map *FunctionsMap* and processed when they are invoked.

```
⟨function-declare_0⟩ ::= ‘@function’ ‘(’ ⟨label_1⟩ ‘,’
                         ‘@out’ ‘:’ ⟨numeric-literal_1⟩ ‘,’
                         ‘@in’ ‘:’ ⟨numeric-literal_2⟩ ’,’
```

```
                    '@instance' ':' ⟨numeric-literal$_3$⟩ ','
                    '@short_witness' ':' ⟨numeric-literal$_4$⟩ ')'
                      ⟨directive-list$_1$⟩
                '@end'
recursion_check {
  process(⟨directive-list$_1$⟩);

  fail_if(FunctionsSet.has(⟨label$_1$⟩));
  functionsSet.put(⟨label$_1$⟩);
}
resource {
  fail_if(!FunctionToggle);
  FunctionsMap.put(⟨label$_1$⟩, {
        outputWireCount: ⟨numeric-literal$_1$⟩.value,
        inputWireCount: ⟨numeric-literal$_2$⟩.value,
        instanceCount: ⟨numeric-literal$_3$⟩.value,
        shortWitnessCount: ⟨numeric-literal$_4$⟩.value,
        body: ⟨directive-list$_1$⟩
      });
}
```

Function invocation is described here. First, it will do some checking to ensure the function exists. Then, it builds a new program *state* with the same size parameters as described by the function's signature. Since attribute processing was postponed by the ⟨function-declare⟩, the invocation processes the cached attribute syntax. Lastly, it will check that all outputs of the function were correctly assigned and that all of the instance and short witnesses were consumed.

```
⟨function-invoke$_0$⟩ ::= [ ⟨wire-list$_1$⟩ '<-' ]? '@call' '('
                 ⟨label$_1$⟩ [ ',' ⟨wire-list$_2$⟩ ]? ')' ';'
recursion_check {
  fail_if(!FunctionsSet.has(⟨label$_1$⟩));
}
resource {
  fail_if(!FunctionToggle);
  fail_if(!FunctionsMap.has(⟨label$_1$⟩));

  function := FunctionsMap.get(⟨label$_1$⟩);

  sub_wires := wireset();
  if(⟨wire-list$_1$⟩) {
    for i from 0 to ⟨wire-list$_1$⟩.indexes.size {
      ⟨function-invoke$_0$⟩.state.wires.remapOutput(
        ⟨wire-list$_1$⟩.indexes[i], sub_wires);
    }
  }
  if(⟨wire-list$_2$⟩) {
    for i from 0 to ⟨wire-list$_2$⟩.indexes.size {
      ⟨function-invoke$_0$⟩.state.wires.remapInput(
        ⟨wire-list$_2$⟩.indexes[i], sub_wires);
    }
  }

  fail_if(sub_wires.outputs.size != function.outputWireCount);
  fail_if(sub_wires.inputs.size != function.inputWireCount);

  function.body.state := state(
    wires := sub_wires,
    iterators := map());

  process(function.body);

  for i from 0 to sub_wires.outputs.size {
    sub_wires.retrieveResource(i);
  }

  fail_if(function.body.instanceCount != function.instanceCount);
  fail_if(function.body.shortWitnessCount != function.shortWitnessCount);
  ⟨function-invoke$_0$⟩.instanceCount := function.instanceCount;
  ⟨function-invoke$_0$⟩.shortWitnessCount := function.shortWitnessCount;
}
evaluation {
  function := FunctionsMap.get(⟨label$_1$⟩;

  sub_instance := stream();
  for i from 0 to function.instanceCount {
    fail_if(⟨function-invoke$_0$⟩.state.instance.size == 0);
    sub_instance.put(⟨function-invoke$_0$⟩.state.instance.pop());
  }
  function.body.state.instance := sub_instance;

  sub_shortWitness := stream();
  for i from 0 to function.shortWitnessCount {
    fail_if(⟨function-invoke$_0$⟩.state.shortWitness.size == 0);
    sub_shortWitness.put(⟨function-invoke$_0$⟩.state.shortWitness.pop());
  }
  function.body.state.shortWitness := sub_shortWitness;
  function.body.state.switchActive :=
    ⟨function-invoke$_.$⟩state.switchActive ;
  process(function.body);

  fail_if(sub_instance.size != 0);
  fail_if(sub_shortWitness.size != 0);
}
```

The ⟨wire-list⟩ is a list of wire indexes composed by both single element items and range of element items. It synthesizes an indexes list attribute.

```
⟨wire-list_0⟩ ::= ⟨wire-list-element_1⟩ [ ‘,’ ⟨wire-list-element_{2...n}⟩ ]*
resource {
  ⟨wire-list_0⟩.indexes := list();
  for i from 1 to n {
    ⟨wire-list_0⟩.indexes.appendAll(⟨wire-list-element_i⟩.indexes);
  }
}

⟨wire-list-element_0⟩ ::= ⟨wire-list-single_1⟩ | ⟨wire-list-range_1⟩
resource {
  ⟨wire-list-element_0⟩.indexes := list();
  match {
    ⟨wire-list-single_1⟩: {
      ⟨wire-list-element_0⟩.indexes.append(⟨wire-list-single_1⟩.index);
    }
    ⟨wire-list-range_1⟩: {
      ⟨wire-list-element_0⟩.indexes.appendAll(⟨wire-list-single_1⟩.indexes);
    }
  }
}

⟨wire-list-single_0⟩ ::= ⟨wire-number_1⟩
resource {
  ⟨wire-list-single_0⟩.index := ⟨wire-number_1⟩.index;
}

⟨wire-list-range_0⟩ ::= ⟨wire-number_1⟩ ‘...’ ⟨wire-number_2⟩
resource {
  fail_if(⟨wire-number_1⟩.index > ⟨wire-number_2⟩.index);

  ⟨wire-list-range_0⟩.indexes := list();

  // interpret loop as being inclusive on both ends
  for i from ⟨wire-number_1⟩.index to ⟨wire-number_2⟩.index {
    ⟨wire-list-range_0⟩.indexes.append(i);
    }
  }
```

The last variety of function-gates are anonymous function gates. Anonymous gates declare their bodies inline, and are invoked immediately. Since they have no name, there is no syntax which could cause them to recurse.

```
  ⟨anon-function_0⟩ ::= [ ⟨wire-list_1⟩ ‘<-’ ]? ‘@anon_call’ ‘(’
                        [ ⟨wire-list_2⟩  ‘,’ ]?
                        ‘@instance’ ‘:’ ⟨numeric-literal_1⟩ ‘,’
                        ‘@short_witness’ ‘:’ ⟨numeric-literal_2⟩ ‘)’
                          ⟨directive-list_1⟩
                        ‘@end’
  resource {
    fail_if(!FunctionToggle);

    sub_wires := wireset();
    if(⟨wire-list_1⟩) {
      for i from 0 to ⟨wire-list_1⟩.indexes.size {
        ⟨anon-function_0⟩.state.wires.remapOutput(
          ⟨wire-list_1⟩.indexes[i], sub_wires);
      }
    }
    if(⟨wire-list_2⟩) {
      for i from 0 to ⟨wire-list_2⟩.indexes.size {
        ⟨anon-function_0⟩.state.wires.remapInput(
          ⟨wire-list_2⟩.indexes[i], sub_wires);
      }
    }

    ⟨directive-list_1⟩.state := state(
      wires := sub_wires,
      iterators := ⟨anon-function_0⟩.state.iterators);

    process(⟨directive-list_1⟩);

    for i from 0 to sub_wires.outputs.size {
      sub_wires.retrieveResource(i);
    }

    fail_if(⟨directive-list_1⟩.instanceCount != ⟨numeric-literal_1⟩.value);
    fail_if(⟨directive-list_1⟩.shortWitnessCount != ⟨numeric-literal_2⟩.value);
    ⟨anon-function_0⟩.instanceCount := ⟨numeric-literal_1⟩.value;
    ⟨anon-function_0⟩.shortWitnessCount := ⟨numeric-literal_2⟩.value;
```

```
}
evaluation {
  sub_instance := stream();
  for i from 0 to ⟨numeric-literal_1⟩.value {
    fail_if(⟨anon-function_0⟩.state.instance.size == 0);
    sub_instance.put(⟨anon-function_0⟩.state.instance.pop());
  }
  ⟨directive-list_1⟩.state.instance := sub_instance;

  sub_shortWitness := stream();
  for i from 0 to ⟨numeric-literal_2⟩.value {
    fail_if(⟨anon-function_0⟩.state.shortWitness.size == 0);
    sub_shortWitness.put(⟨anon-function_0⟩.state.shortWitness.pop());
  }
  ⟨directive-list_1⟩.state.shortWitness := sub_shortWitness;
  ⟨directive-list_1⟩.state.switchActive :=
    ⟨anon-function_.⟩state.switchActive;

  process(⟨directive-list_1⟩);

  fail_if(sub_instance.size != 0);
  fail_if(sub_shortWitness.size != 0);
}
```

4.7 Relation: For Loops

A for-loop simply repeats a function a set number of times. To accomplish this in the attribute grammar we introduce functionality for "clear attributes" and "reprocess attributes". We also need to introduce syntax for altering the function's inputs and outputs on each iteration. Iterator expressions will be described in detail, but briefly they replace wire indexes in the ⟨wire-list⟩ with expressions over loop iterators.

```
⟨for-loop_0⟩ ::= [ ⟨wire-list_1⟩ '<-' ]? '@for' ⟨label_1⟩
             '@first' ⟨numeric-literal_1⟩ '@last' ⟨numeric-literal_2⟩
               [ ⟨iter-expr-function-invoke_1⟩ | ⟨iter-expr-anon-function_1⟩ ]
             '@end'
resource {
  fail_if(!ForLoopToggle);
  fail_if(⟨numeric-literal_1⟩ > ⟨numeric-literal_1⟩);

  fail_if(⟨for-loop_0⟩.state.iterators.has(⟨label_1⟩));
  ⟨for-loop_0⟩.state.iterators.put(⟨label_1⟩, 0);
  loopOutputs := list();
  if(⟨wire-list$_1$⟩) {
    loopOutputs := ⟨wire-list$_1$⟩.indexes;
  }

  ⟨for-loop$_0$⟩.instanceCount := 0;
  ⟨for-loop$_0$⟩.shortWitnessCount := 0;
}

// loop is interpreted as inclusive on both ends
for i from ⟨numeric-literal$_1$⟩.value to ⟨numeric-literal$_2$⟩.value {
  resource {
    ⟨for-loop$_0$⟩.state.iterators.set(⟨label$_1$⟩, i);
    match {
      ⟨iter-expr-function-invoke$_1$⟩: {
        ⟨iter-expr-function-invoke$_1$⟩.state := ⟨for-loop$_9$⟩.state;
        ⟨iter-expr-function-invoke$_1$⟩.loopOutputs := loopOutputs;

        ⟨for-loop$_0$⟩.instanceCount +=
          ⟨iter-expr-function-invoke$_1$⟩.instanceCount;
        ⟨for-loop$_0$⟩.shortWitnessCount +=
          ⟨iter-expr-function-invoke$_1$⟩.shortWitnessCount;
      }
      ⟨iter-expr-anon-function$_1$⟩: {
        ⟨iter-expr-anon-function$_1$⟩.state := ⟨for-loop$_9$⟩.state;
        ⟨iter-expr-function-invoke$_1$⟩.loopOutputs := loopOutputs;

        ⟨for-loop$_0$⟩.instanceCount +=
          ⟨iter-expr-anon-function$_1$⟩.instanceCount;
        ⟨for-loop$_0$⟩.shortWitnessCount +=
          ⟨iter-expr-anon-function$_1$⟩.shortWitnessCount;
      }
    }
  }
  evaluation {
    match {
      ⟨iter-expr-function-invoke$_1$⟩: {
        process(⟨iter-expr-function-invoke$_1$⟩);
      }
      ⟨iter-expr-anon-function$_1$⟩: {
        process(⟨iter-expr-anon-function$_1$⟩);
      }
    }
  }
}
```

```
resource {
  ⟨for-loop_0⟩.state.iterators.remove(⟨label_1⟩);

  for i from 0 to loopOutputs.size {
    ⟨for-loop_0⟩.state.wires.retrieveResource(loopOutputs[i]);
  }
}
```

In order to meaningfully repeat the function however, the inputs and outputs need to advance on each iteration. To do this, we introduce "iterator expressions" as a special syntax replacing the ⟨wire-list⟩ used by function invocations. An iterator expression takes the place of each ⟨wire-number⟩ element within the ⟨wire-list⟩. The ⟨iter-expr-function-invoke⟩ and ⟨iter-expr-anon-function⟩ will use iterator expressions within their ⟨wire-list⟩s to express advancement on an interation. The ⟨iter-expr-wire-list⟩, shown later, has the same form as a ⟨wire-list⟩, but with ⟨iter-expr-wire-number⟩s instead of regular ⟨wire-numbers.⟩

Iterator expressions require the *iterators* object of the *state*. Their arithmetic uses integers in the range 0 through $2^{64} - 1$ with wraparound. The *iterators* map is updated using lexical scoping rules. Briefly, when entering an anonymous function, containing loop iterators are transferred into the new scope, but when entering named functions, they are not.

Note that ⟨iter-expr-wire-number⟩ is slightly ambiguous with a ⟨wire-number⟩, as $123 could be interpreted either way. Interpret this as to allow promotion from a ⟨wire-number⟩ to a ⟨iter-expr-wire-number⟩, where the latter is required.

```
⟨iter-expr-wire-number_0⟩ ::= '$' ⟨iter-expr_1⟩
resource {
  ⟨iter-expr_1⟩.iterators := ⟨iter-expr-wire-number_0⟩.state.iterators;
  ⟨iter-expr-wire-number_0⟩.index := ⟨iter-expr_1⟩.value;
}

⟨iter-expr_0⟩ ::= ⟨numeric-literal_1⟩
resource {
  ⟨iter-expr_0⟩.value := ⟨numeric-literal_1⟩.value;
}

⟨iter-expr_0⟩ ::= ⟨label_1⟩
resource {
  fail_if(!⟨iter-expr_0⟩.iterators.has(⟨label_1⟩));
  ⟨iter-expr_0⟩.value := ⟨iter-expr_0⟩.iterators.get(⟨label_1⟩);
}

⟨iter-expr_0⟩ ::= '(' ⟨iter-expr_1⟩ '+'  ⟨iter-expr_2⟩ ')'
```

```
  resource {
    ⟨iter-expr$_1$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    ⟨iter-expr$_2$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    process(⟨iter-expr$_{1...2}$⟩);
    ⟨iter-expr$_0$⟩.value := ⟨iter-expr$_1$⟩.value + ⟨iter-expr$_2$⟩.value;
  }

  ⟨iter-expr$_0$⟩ ::= ‘(’ ⟨iter-expr$_1$⟩ ‘-’  ⟨iter-expr$_2$⟩ ‘)’
  resource {
    ⟨iter-expr$_1$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    ⟨iter-expr$_2$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    process(⟨iter-expr$_{1...2}$⟩);
    ⟨iter-expr$_0$⟩.value := ⟨iter-expr$_1$⟩.value - ⟨iter-expr$_2$⟩.value;
  }

  ⟨iter-expr$_0$⟩ ::= ‘(’ ⟨iter-expr$_1$⟩ ‘*’  ⟨iter-expr$_2$⟩ ‘)’
  resource {
    ⟨iter-expr$_1$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    ⟨iter-expr$_2$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    process(⟨iter-expr$_{1...2}$⟩);
    ⟨iter-expr$_0$⟩.value := ⟨iter-expr$_1$⟩.value * ⟨iter-expr$_2$⟩.value;
  }
```

In addition to sum, difference, and product expressions, an ⟨iter-expr⟩ may include division by a numeric constant. This computes the integer quotient. The numerator may be an arbitrary expression, including other divisions, and the denominator a non-zero constant. Numerator, denominator, and quotient values are, of course, all 64-bit unsigned integers with wraparound.

```
  ⟨iter-expr$_0$⟩ ::= ‘(’ ⟨iter-expr$_1$⟩ ‘/’  ⟨numeric-literal$_1$⟩ ‘)’
  resource {
    fail_if(⟨numeric-literal$_1$⟩.value == 0);
    ⟨iter-expr$_1$⟩.iterators := ⟨iter-expr$_0$⟩.iterators;
    process(⟨iter-expr$_1$⟩);
    ⟨iter-expr$_0$⟩.value := ⟨iter-expr$_1$⟩.value * ⟨numeric-literal$_1$⟩.value;
  }
```

The ⟨iter-expr-wire-list⟩ will mirror the form of a ⟨wire-list⟩, by replacing each of the ⟨wire-number⟩s with the calculated value of a ⟨iter-expr-wire-number⟩.

```
  ⟨iter-expr-wire-list$_0$⟩ ::= ⟨iter-expr-wire-element$_1$⟩
                          [ ‘,’ ⟨iter-expr-wire-element$_{2...n}$⟩ ]*
  resource {
    ⟨iter-expr-wire-list$_0$⟩.indexes := list();
    for i from 1 to n {
```

```
    ⟨iter-expr-wire-list-element$_i$⟩.iterators :=
      ⟨iter-expr-wire-list$_0$⟩.iterators;
    process(⟨iter-expr-wire-list-element$_i$⟩);
    ⟨iter-expr-wire-list$_0$⟩.appendAll(
      ⟨iter-expr-wire-list-element$_i$⟩.indexes);
  }
}

⟨iter-expr-wire-list-element$_0$⟩ ::= ⟨iter-expr-wire-list-single$_1$⟩
                                  | ⟨iter-expr-wire-list-range$_1$⟩
resource {
  ⟨iter-expr-wire-list-element$_0$⟩.indexes := list();
  match {
    ⟨iter-expr-wire-list-single$_1$⟩: {
      ⟨iter-expr-wire-list-single$_1$⟩.iterators :=
        ⟨iter-expr-wire-list-element$_0$⟩.iterators;
      process(⟨iter-expr-wire-list-single$_1$⟩);
      ⟨iter-expr-wire-list-element$_0$⟩.indexes.append(
        ⟨iter-expr-wire-list-single$_1$⟩.index);
    }
    ⟨iter-expr-wire-list-range$_1$⟩: {
      ⟨iter-expr-wire-list-range$_1$⟩.iterators :=
        ⟨iter-expr-wire-list-element$_0$⟩.iterators;
      process(⟨iter-expr-wire-list-range$_1$⟩);
      ⟨iter-expr-wire-list-element$_0$⟩.indexes.appendAll(
        ⟨iter-expr-wire-list-range$_1$⟩.indexes);
    }
  }
}

⟨iter-expr-wire-list-single$_0$⟩ ::= ⟨iter-expr-wire-number$_1$⟩
resource {
  ⟨iter-expr-wire-number$_1$⟩.iterators :=
    ⟨iter-expr-wire-list-single$_0$⟩.iterators;
  processs(⟨iter-expr-wire-number$_1$⟩);
  ⟨iter-expr-wire-list-single$_0$⟩.index :=
    ⟨iter-expr-wire-number$_1$⟩.index;
}

⟨iter-expr-wire-list-range$_0$⟩ ::= ⟨iter-expr-wire-number$_1$⟩
                                 ‘...’ ⟨iter-expr-wire-number$_2$⟩
resource {
  ⟨iter-expr-wire-number$_1$⟩.iterators :=
    ⟨iter-expr-wire-list-range$_0$⟩.iterators;
  ⟨iter-expr-wire-number$_2$⟩.iterators :=
    ⟨iter-expr-wire-list-range_0⟩.iterators;

  processs(⟨iter-expr-wire-number_{1...2}⟩);
  fail_if(⟨iter-expr-wire-number_1⟩.index > ⟨iter-expr-wire-number_2⟩.index);

  ⟨iter-expr-wire-list-range_0⟩.indexes := list();

  // interpret loop as being inclusive on both ends
  for i from ⟨iter-expr-wire-number_1⟩.index
      to ⟨iter-expr-wire-number_2⟩.index {
    ⟨iter-expr-wire-list-range_0⟩.indexes.append(i);
  }
}
```

Finally, the ⟨iter-expr-function-invoke⟩ and ⟨iter-expr-anon-function⟩ take the place of ⟨function-invoke⟩ and ⟨anon-function⟩ with the added iterator expression functionality. Their semantics are largely equivalent, except for the addition of checking output wires for set membership in the for loop's output list – given as an inherited attribute, loopOutputs.

```
⟨iter-expr-function-invoke_0⟩ ::= [ ⟨iter-expr-wire-list_1⟩ ‘<-’ ]?
                                 ‘@call’ ‘(’ ⟨label_1⟩
                                 [ ‘,’ ⟨iter-expr-wire-list_2⟩ ]? ‘)’ ‘;’
recursion_check {
  fail_if(!FunctionsSet.has(⟨label_1⟩));
}
resource {
  fail_if(!FunctionToggle);
  fail_if(!FunctionsMap.has(⟨label_1⟩));

  function := FunctionsMap.get(⟨label_1⟩);

  sub_wires := wireset();
  if(⟨iter-expr-wire-list_1⟩) {
    ⟨iter-expr-wire-list_1⟩.iterators :=
      ⟨iter-expr-function-invoke_0⟩.state.iterators;
    process(⟨iter-expr-wire-list_1⟩);

    for i from 0 to ⟨iter-expr-wire-list_1⟩.indexes.size {
      fail_if(!⟨iter-expr-function-invoke_0⟩.loopOutputs.has(
        ⟨iter-expr-wire-list_1⟩.indexes[i]));

      ⟨iter-expr-function-invoke_0⟩.state.wires.remapOutput(
        ⟨iter-expr-wire-list_1⟩.indexes[i], sub_wires);
    }
  }
  if(⟨iter-expr-wire-list$_2$⟩) {
    ⟨iter-expr-wire-list$_2$⟩.iterators :=
      ⟨iter-expr-function-invoke$_0$⟩.state.iterators;
    process(⟨iter-expr-wire-list$_2$⟩);

    for i from 0 to ⟨iter-expr-wire-list$_2$⟩.indexes.size {
      ⟨iter-expr-function-invoke$_0$⟩.state.wires.remapInput(
        ⟨iter-expr-wire-list$_2$⟩.indexes[i], sub_wires);
    }
  }

  fail_if(sub_wires.outputs.size != function.outputWireCount);
  fail_if(sub_wires.inputs.size != function.inputWireCount);

  function.body.state := state(
    wires := sub_wires,
    iterators := map());

  process(function.body);

  for i from 0 to sub_wires.outputs.size {
    sub_wires.retrieveResource(i);
  }

  fail_if(function.body.instanceCount != function.instanceCount);
  fail_if(function.body.shortWitnessCount != function.shortWitnessCount);
  ⟨iter-expr-function-invoke$_0$⟩.instanceCount := function.instanceCount;
  ⟨iter-expr-function-invoke$_0$⟩.shortWitnessCount :=
    function.shortWitnessCount;
}
evaluation {
  function := FunctionsMap.get(⟨label$_1$⟩);

  sub_instance := stream();
  for i from 0 to function.instancCount {
    fail_if(⟨iter-expr-function-invoke$_0$⟩.state.instance.size == 0);
    sub_instance.put(⟨iter-expr-function-invoke$_0$⟩.state.instance.pop());
  }
  function.body.state.instance := sub_instance;

  sub_shortWitness := stream();
  for i from 0 to function.shortWitnessCount {
    fail_if(⟨iter-expr-function-invoke$_0$⟩.state.shortWitness.size == 0);
    sub_shortWitness.put(
      ⟨iter-expr-function-invoke_0⟩.state.shortWitness.pop());
  }
  function.body.state.shortWitness := sub_shortWitness;
  function.body.state.switchActive :=
    ⟨iter-expr-function-invoke.⟩state.switchActive;

  process(function.body);

  fail_if(sub_instance.size != 0);
  fail_if(sub_shortWitness.size != 0);
}

⟨iter-expr-anon-function_0⟩ ::= [ ⟨iter-expr-wire-list_1⟩ '<-' ]?
                            '@anon_call' '('
                            [ ⟨iter-expr-wire-list_2⟩ ',' ]?
                            '@instance' ':' ⟨numeric-literal_1⟩ ','
                            '@short_witness' ':' ⟨numeric-literal_2⟩ ')'
                              ⟨directive-list_1⟩
                            '@end'
resource {
  sub_wires := wireset();
  if(⟨iter-expr-wire-list_1⟩) {
    ⟨iter-expr-wire-list_1⟩.iterators :=
      ⟨iter-expr-anon-function_0⟩.state.iterators;
    process(⟨iter-expr-wire-list_1⟩);

    for i from 0 to ⟨iter-expr-wire-list_1⟩.indexes.size {
      fail_if(!⟨iter-expr-anon-function_0⟩.loopOutputs.has(
        ⟨iter-expr-wire-list_1⟩.indexes[i]));

      ⟨iter-expr-anon-function_0⟩.state.wires.remapOutput(
        ⟨iter-expr-wire-list_1⟩.indexes[i], sub_wires);
    }
  }
  if(⟨iter-expr-wire-list_2⟩) {
    ⟨iter-expr-wire-list_2⟩.iterators :=
      ⟨iter-expr-anon-function_0⟩.state.iterators;
    process(⟨iter-expr-wire-list_1⟩);

    for i from 0 to ⟨iter-expr-wire-list_2⟩.indexes.size {
      ⟨iter-expr-anon-function_0⟩.state.wires.remapInput(
        ⟨iter-expr-wire-list_2⟩.indexes[i], sub_wires);
    }
  }
```

```
    ⟨directive-list$_1$⟩.state := state(
      wires := sub_wires,
      iterators := map());

    process(⟨directive-list$_1$⟩);

    for i from 0 to sub_wires.outputs.size {
      sub_wires.retrieveResource(i);
    }

    fail_if(⟨directive-list$_1$⟩.instanceCount != ⟨numeric-literal$_1$⟩.value);
    fail_if(⟨directive-list$_1$⟩.shortWitnessCount != ⟨numeric-literal$_2$⟩.value);
    ⟨iter-expr-anon-function$_0$⟩.instanceCount := ⟨numeric-literal$_1$⟩.value;
    ⟨iter-expr-anon-function$_0$⟩.shortWitnessCount := ⟨numeric-literal$_2$⟩.value;
  }
  evaluation {
    sub_instance := stream();
    for i from 0 to function.instanceWireCount {
      fail_if(⟨iter-expr-anon-function$_0$⟩.state.instance.size == 0);
      sub_instance.put(⟨iter-expr-anon-function$_0$⟩.state.instance.pop());
    }
    ⟨directive-list$_1$⟩.state.instance := sub_instance;

    sub_shortWitness := stream();
    for i from 0 to function.shortWitnessWireCount {
      fail_if(⟨iter-expr-anon-function$_0$⟩.state.shortWitness.size == 0);
      sub_shortWitness.put(
        ⟨iter-expr-anon-function$_0$⟩.state.shortWitness.pop());
    }
    ⟨directive-list$_1$⟩.state.shortWitness := sub_shortWitness;
    ⟨directive-list$_1$⟩.state.switchActive :=
      ⟨iter-expr-anon-function$_0$⟩.state.switchActive;

    process(⟨directive-list$_1$⟩);

    fail_if(sub_instance.size != 0);
    fail_if(sub_shortWitness.size != 0);
  }
```

4.8 Relation: Switch-Case Statements

The switch-case emulates branching in a circuit by selecting the correct result and discarding the remaining results. Concretely, each case will assign "dummy outputs", and during evaluation, when a case is selected, its "dummy" outputs are copied onto the actual output wires.

It also has checks to assure that each case is unique and that the output wires are assignable during the *resource validity* phase. During the *evaluation validity* phase, it checks that one of the cases is indeed selected. For instance and short witness streams, the reference behavior is to duplicate the streams, demonstrating that a case-statement consumes as many values as the maximum of its case implementations. There should exist "stream rewind" and "input wire rewrite" (outlined in section 6.3) techniques to achieve the same semantic.

```
⟨switch-statement$_0$⟩ ::= [ ⟨wire-list$_1$⟩ ‘<-’ ]?
                            ‘@switch’ ‘(’ ⟨wire-number$_1$⟩ ‘)’
                            [ ‘@case’ ⟨field-literal$_{1...n}$⟩ ‘:’
                              ⟨case-function$_{1...n}$⟩
                            ]+
                            ‘@end’
resource {
  fail_if(!SwitchCaseToggle);

  ⟨switch-statement$_0$⟩.state.wires.retrieveResource(⟨wire-number$_1$⟩.index);

  for i from 0 to ⟨wire-list$_1$⟩.indexes.size {
    ⟨switch-statement$_0$⟩.state.wires.insertResource(
      ⟨wire-list$_1$⟩.indexes[i]);
  }

  caseSelectors = set();
  for i from 1 to n {
    fail_if(⟨field-literal$_i$⟩.value >= p);
    fail_if(caseSelectors.has(⟨field-literal$_i$⟩.value));
    caseSelectors.put(⟨field-literal$_i$⟩.value)

    ⟨case-function$_i$⟩.state := ⟨switch-statement$_0$⟩.state;
    ⟨case-function$_i$⟩.outputSize = ⟨wire-list$_1$⟩.indexes.size;

    process(⟨case-function$_i$⟩);
  }

  ⟨switch-statement$_0$⟩.instanceCount :=
    maximum(⟨case-function$_{1...n}$⟩.instanceCount);
  ⟨switch-statement$_0$⟩.shortWitnessCount :=
    maximum(⟨case-function$_{1...n}$⟩.instanceCount);
}
evaluation {
  instance_dup := list();
  for i from 0 to ⟨switch-statement$_0$⟩.instanceCount {
```

```
  instance_dup.append(
    ⟨switch-statement$_0$⟩.state.instance.pop());
}

shortWitness_dup := list();
for i from 0 to ⟨switch-statement$_0$⟩.shortWitnessCount {
  shortWitness_dup.append(
    ⟨switch-statement$_0$⟩.state.shortWitness.pop());
}

matchedCase = false;
selectValue = ⟨switch-statement$_0$⟩.state.wires.retrieveEvaluation(
  ⟨wire-number$_1$⟩.value);

for i from 1 to n {
  sub_instance := stream();
  for j from 0 to ⟨case-function$_i$⟩.instanceCount {
    sub_instance.push(instance_dup[j]);
  }

  sub_shortWitenss := stream();
  for j from 0 to ⟨case-function$_i$⟩.shortWitnessCount {
    sub_shortWitness.push(shortWitness_dup[j]);
  }

  ⟨case-function$_i$⟩.sub_instance := sub_instance;
  ⟨case-function$_i$⟩.sub_shortWitness := sub_shortWitness;

  if(selectValue == ⟨field-literal$_i$⟩.value) {
    matchedCase := true;
    ⟨case-function$_i$⟩.sub_switchActive :=
      ⟨switch-statement$_0$⟩.state.switchActive;

    process(⟨case-function$_i$⟩);

    for j from 0 to ⟨wire-list$_1$⟩.size {
      ⟨switch-statement$_0$⟩.state.wires.insertEvaluation(
        ⟨wire-list$_1$⟩.indexes[j], ⟨case-function$_i$⟩.outputs[j]);
    }
  }
  else {
    ⟨case-function$_i$⟩.sub_switchActive := false;
    process(⟨case-function$_i$⟩);
  }
}
```

```
  fail_if(⟨switch-statement_0⟩.state.switchActive && !matchedCase);
}

⟨case-function_0⟩ ::= ⟨case-function-invoke_1⟩ | ⟨case-anon-function_1⟩
resource {
  match {
    ⟨case-function-invoke_1⟩: {
      ⟨case-function-invoke_1⟩.state := ⟨case-function_0⟩.state;
      ⟨case-function-invoke_1⟩.outputSize := ⟨case-function_0⟩.outputSize;

      process(⟨case-function-invoke_1⟩);

      ⟨case-function_0⟩.instanceCount :=
        ⟨case-function-invoke_1⟩.instanceCount;
      ⟨case-function_0⟩.shortWitnessCount :=
        ⟨case-function-invoke_1⟩.shortWitnessCount;
    }
    ⟨case-anon-function_1⟩: {
      ⟨case-anon-function_1⟩.state := ⟨case-function_0⟩.state;
      ⟨case-anon-function_1⟩.outputSize := ⟨case-function_0⟩.outputSize;

      process(⟨case-anon-function_1⟩);

      ⟨case-function_0⟩.instanceCount :=
        ⟨case-anon-function_1⟩.instanceCount;
      ⟨case-function_0⟩.shortWitnessCount :=
        ⟨case-anon-function_1⟩.shortWitnessCount;
    }
  }
}
evaluation {
  match {
    ⟨case-function-invoke_1⟩: {
      ⟨case-function-invoke_1⟩.sub_instance :=
        ⟨case-function_0⟩.sub_instance;
      ⟨case-function-invoke_1⟩.sub_shortWitness :=
        ⟨case-function_0⟩.sub_shortWitness;
      ⟨case-function-invoke_1⟩.sub_switchActive :=
        ⟨case-function_0⟩.sub_switchActive;

      process(⟨case-function-invoke_1⟩);

      ⟨case-function_0⟩.outputs :=
        ⟨case-anon-function_1⟩.outputs;
    }
    ⟨case-anon-function$_1$⟩: {
      ⟨case-anon-function$_1$⟩.sub_instance :=
        ⟨case-function$_0$⟩.sub_instance;
      ⟨case-anon-function$_1$⟩.sub_shortWitness :=
        ⟨case-function$_0$⟩.sub_shortWitness;
      ⟨case-anon-function$_1$⟩.sub_switchActive :=
        ⟨case-function$_0$⟩.sub_switchActive;

      process(⟨case-function-invoke$_1$⟩);

      ⟨case-function$_0$⟩.outputs :=
        ⟨case-anon-function$_1$⟩.outputs;
    }
  }
}

⟨case-function-invoke$_0$⟩ ::= ‘@call’ ‘(’ ⟨label$_1$⟩
                         [ ‘,’ ⟨wire-list$_1$⟩ ]? ‘)’ ‘;’
recursion_check {
  fail_if(!functionsSet.has(⟨label$_1$⟩));
}
resource {
  fail_if(!FunctionToggle);
  fail_if(!functionsMap.has(⟨label$_1$⟩));

  function := functionsMap.get(⟨label$_1$⟩);

  fail_if(function.outputWireCount != ⟨case-function-invoke$_.$⟩outputSize);

  sub_wires := wireset();
  sub_wires.mapDummies(⟨case-function-invoke$_0$⟩.outputSize);

  if(⟨wire-list$_1$⟩) {
    for i from 0 to ⟨wire-list$_1$⟩.indexes.size {
      ⟨case-function-invoke$_0$⟩.state.wires.remapInput(
        ⟨wire-list$_1$⟩.indexes[i], sub_wires);
    }
  }

  fail_if(sub_wires.inputs.size != function.inputWireCount);

  function.body.state := state(
    wires := sub_wires,
    iterators := map());
  process(function.body);

  for i from 0 to sub_wires.outputs.size {
    sub_wires.retrieveResource(i);
  }

  fail_if(function.body.instanceCount != function.instanceCount);
  fail_if(function.body.shortWitnessCount != function.shortWitnessCount);
  ⟨case-function-invoke$_0$⟩.instanceCount := function.instanceCount;
  ⟨case-function-invoke$_0$⟩.shortWitnessCount := function.shortWitnessCount;
}
evaluation {
  function := functionsMap.get(⟨label$_1$⟩);

  function.body.state.instance := ⟨case-function-invoke$_0$⟩.sub_instance;
  function.body.state.shortWitness :=
    ⟨case-function-invoke$_0$⟩.sub_shortWitness;
  function.body.state.switchActive :=
    ⟨case-function-invoke$_0$⟩.sub_switchActive;

  process(function.body);

  fail_if(⟨case-function-invoke$_0$⟩.sub_instance.size != 0);
  fail_if(⟨case-function-invoke$_0$⟩.sub_shortWitness.size != 0);

  ⟨case-function-invoke$_0$⟩.outputs := function.body.state.wires.outputs;
}

⟨case-anon-function$_0$⟩ ::= ‘@anon_call’ ‘(’
                         [ ⟨wire-list$_1$⟩ ‘,’ ]?
                         ‘@instance’ ‘:’  ⟨numeric-literal$_1$⟩ ‘,’
                         ‘@short_witness’ ‘:’ ⟨numeric-literal$_2$⟩ ‘)’
                           ⟨directive-list$_1$⟩
                         ‘@end’
resource {
  sub_wires := wireset();
  sub_wires.mapDummies(⟨case-anon-function$_0$⟩.outputSize);

  if(⟨wire-list$_1$⟩) {
    for i from 0 to ⟨wire-list$_1$⟩.indexes.size {
      ⟨case-anon-function$_0$⟩.state.wires.remapInput(
        ⟨wire-list$_1$⟩.indexes[i], sub_wires);
    }
  }
  ⟨directive-list$_1$⟩.state := state(
    wires := sub_wires,
    iterators := map());

  process(⟨directive-list$_1$⟩);

  for i from 0 to sub_wires.outputs.size {
    sub_wires.retrieveResource(i);
  }

  fail_if(⟨directive-list$_1$⟩.instanceCount != ⟨numeric-literal$_1$⟩.value);
  fail_if(⟨directive-list$_1$⟩.shortWitnessCount != ⟨numeric-literal$_2$⟩.value);
  ⟨case-anon-function$_0$⟩.instanceCount := ⟨numeric-literal$_1$⟩.value;
  ⟨case-anon-function$_0$⟩.shortWitnessCount := ⟨numeric-literal$_1$⟩.value;
}
evaluation {
  ⟨directive-list$_1$⟩.state.instance :=
    ⟨case-function-invoke$_0$⟩.sub_instance;
  ⟨directive-list$_1$⟩.state.shortWitness :=
    ⟨case-anon-function$_0$⟩.sub_shortWitness;
    ⟨directive-list$_1$⟩.state.switchActive :=
    ⟨case-anon-function$_0$⟩.sub_switchActive;

  process(⟨directive-list$_1$⟩);

  fail_if(⟨case-anon-function$_0$⟩.sub_instance.size != 0);
  fail_if(⟨case-anon-function$_0$⟩.sub_shortWitness.size != 0);

  ⟨case-anon-function$_0$⟩.outputs := ⟨directive-list$_1$⟩.state.wires.outputs;
}
```

5 Binary Serialization (FlatBuffers)

The binary serialization of IR0 will be described here using the open-source FlatBuffers cross-platform serialization library, originally developed by Google. FlatBuffers is a metaformat that specifies the superficial aspects of the syntax, such as representations of literals, structured data and arrays. It moreover supports formal schemas that concretely define what elements (e.g., structures and arrays) can appear in the specific format. FlatBuffers was chosen for the following reasons:

- It offers an existing compact encoding of the format with efficient (de)serialization.
- It is supported by a wide-range of community-based tools and libraries for the most common languages (and is also easy to parse from scratch).

The concrete FlatBuffers schema, which implements the abstract semantics summarized in Section 3. Each element of the FlatBuffer schema elements is described as isomorphic to the textual syntax in Section 4, establishing a clear equivalence to the text format and the IR semantics.

Before looking at the specific messages and their syntax, let's take a look at how circuits are interpreted in the FlatBuffers schema. Circuits are represented by following the simple design philosophy that "every variable is the input or output of a gate". Wires are defined as the main object of the schema, with wires specifying the connections between the gates such that each wire can be the input to or output of a gate. In most cases, it is both: the output of a preceding gate that becomes the input to the following connected gate. Wires can be reused across several gates, forming a topology. Variables are then assigned to wires as concrete values.

As we will see in Section 5.4.2, to make the schema simpler to use, we made it so that different variable types that are inputs to the circuit (instance, witness and constant) will be used by assigning them to the output wire of a gate of that type (i.e.: constant variables are the output of a constant gate).

When dealing with binaries and files in FlatBuffers, we specify a file extension and identification constant. The constant is present at the beginning of all binary, FlatBuffers-based, SIEVE IR resource files or streams, ensuring the user is reading the kind of file they are expecting. These are defined as such

```
    // When storing messages to files, this extension and
    // identifier should be used.
    file_extension "sieve";
    file_identifier "siev"; // constant bytes for file identification
```

In general the attributes of a FlatBuffer `table` element are allowed to be `null` values. In the IR, except where noted, a null value is considered a *syntax invalidity*.

Resources as FBS Messages

As per the abstract syntax in Section 3, there are three message types that can be communicated through files or streams: Instance, Witness and Relation. When streaming messages, the producer will always include the Header as part of the messages in order to transmit the semantics to the consumer. For this schema, the messages should be framed such that all messages must be prefixed by the size of the message, not including the prefix, as a 4-bytes little-endian unsigned integer.

```
union Message {
   Relation,
   Instance,
   Witness,
}
```

FlatBuffers allows you to parse the structures and messages from a concretely defined starting point, called the root_type. We implemented it through the Root struct that calls a message.

Due to 32-bit internal pointers, FlatBuffers have a 2GB limit on message size. Because of this we prefix each Root element with a 32-bit unsigned little-endian constant describing the length of the Root in bytes. These length, Root pairs may be repeated as many times as necessary so long as each Root has the same message type.

```
// All message types are encapsulated in the FlatBuffers root table
   .
table Root {
    message                    :Message;
}
root_type Root;
```

In order to maintain isomorphism with the text format, a few constraints are added to the *syntactic validity* of such sequenced FlatBuffers.

- The header, gate set and features of each must be identical.
- The text syntax lists all function declarations ahead of directives in the top-level scope. A FlatBuffer in sequence may not list additional function declarations if a prior FlatBuffer of the sequence already listed directives of the top-level scope.

Semantics in FlatBuffers. As a structured format, the FlatBuffers schema provides a concrete, readable and typed syntax, ensuring part of the semantics of the IR around the types of the fields. However, it does not provide resource or evaluation validity as it is not a language. As explained in Section 1.2, there are three validation methods that need to be defined: *syntactic validity*, which is mostly ensured by the structures and types of the schema provided; *resource validity* and *evaluation validity*, both of which are defined in section 4 and summarized in this section. If definitions from section 4 differ from summaries in this section, then section 4 takes precedence.

The validation of the circuit should never modify it, and should consider the circuit and its topology as a read-only parameter. In this section we provide both the syntax for the FlatBuffers schema, and in each subsection, a review of the semantic checks to be performed by the *resource validity* and *evaluation validity* checks.

5.1 Messages, Abstractions and Basic Types

The four basic types in the FlatBuffers IR schema are

- A Directive defines a concrete computation in the circuit
- A Wire is a unique identifier and defines the flow of values through the computation
- A Value is a field element and is to be assigned to a specific Wire.
- An IterExprWireNumber is a type used uniquely within *for loops* to allow computing expressions on values that differ per iteration of the loop. We call these *iterators*.

Concretely, a Directive is the principal object of the schema, defining the concrete computation or directive on variables, and by extension, on value assignment of variables. A gate is defined by the List of Directives, as described in Section 5.4. Note that the Directive in concert with DirectiveSet form an isomorphism to ⟨directive⟩ from section 4.4.

```
table Directive {
    directive      :DirectiveSet;
}
```

A relation is defined by connecting directives in some topological order through the use of *wires*. Each Wire is referenced by a unique identifier, `id`, which can be assigned or empty (also after being freed). Every wire is the output of a gate and hence every variable can be seen as a wire. Note that the `Wire` table is equivalent to the ⟨wire-number⟩ element defined by appendix A

```
table Wire {
    id         :uint64;
}
```

In some cases, a reference to several consecutive wires is needed, so we use a range of identifiers as a `WireRange`. Note that the `WireRange` table is isomorphic to the ⟨wire-range⟩ element of section 4.6.

```
table WireRange {
    first      :Wire;
    last       :Wire;
}
```

In order to allow for any of the two types above, we build this union trait, as is common in FlatBuffers. It is equivalent, in combination with table `WireListElement`, to the ⟨wire-list-element⟩ from section 4.6.

```
union WireListElementU {
    Wire,
    WireRange,
}
```

Currently, the Rust implementation of Flatbuffers does not allow creation of an array structure from a *union* of elements, hence we need to invoke the following trick: to wrap a the union with a table, as seen below. The WireList forms a compact list in which elements can be single wires or ranges of wires. the WireList is isomorphic to ⟨wire-list⟩ of section 4.6. References to a `WireList` may be empty, but must not be `null`.

```
table WireListElement {
    element   :WireListElementU;
}

table WireList {
    elements    :[WireListElement];
}
```

Later we will see directives which read values from input streams and assign their values to a specific wire. These input streams are sequences of input values, each of which is read sequentially. The type Value represents a field element as a vector of bytes, encoded least significant byte first. The `Value` is isomorphic to the ⟨`field-literal`⟩ of appendix A. A `Value` may have leading zeros, at the discretion of the IR producer.

```
table Value {
    value        :[ubyte];
}
```

Header Syntax & Semantics

Header Syntax. The Header struct, included in all message types, contains the basic information of the semantics of the witnessed statement. The Header table is isomorphic to the ⟨`header`⟩ element of section 4.1. Specifically, it contains

- The `version`, of type string, specifies the concrete version of the IR being used and is formatted as a string based on the semantic versioning system explained in Section 3. In the FlatBuffers schema, this is for reference between the consumer and producer of the witnessed statement, as there is native backwards and forward compatibility with the formal schema.

- The `field_characteristic`, of type Value, which defines the underlying field modulus of the statement. Values in the vector `value` must be represented under this characteristic.

- The `field_degree`, which determines the extension degree of the field used. For IR v1.0.0 and before, the only allowed parameter value for the field extension is one (1). The parameter is added for the purpose of extensibility in future versions. The `field_characteristic` is encoded least significant byte first.

```
table Header {
   version              :string;
   field_characteristic :Value;
   field_degree         :uint32;
}
```

Header Semantics. The following is a comprehensive list of the semantic checks to be performed around the header of the IR.

- *Versioning:* ensure that the version string has the correct format (e.g. matches the following regular expression “`^\d+.\d+.\d+$`”) as defined in the semantic versioning in Section 3.1.

- *Field characteristic:* ensure that the characteristic is strictly greater than 1 and is a prime.
- *Field degree:* ensure that the field degree is exactly 1.
- *Consistency:* as part of the *evaluation validity*, ensure that the header within messages are coherent for a given circuit: (a) versions should be identical; (2) the field characteristic and the field degree should be the same.

5.2 Inputs: Instance & Witness

Instance Syntax. The *Instance* message transmits the assignments of the common inputs to the circuit, known by the prover and the verifier. The Instance table is isomorphic to the ⟨instance⟩ element from section 4.2.

- The header, of type Header, is included for communicating the semantics between a producer and a consumer.
- The common_inputs, a vector of type Value, specify the public values assigned to wires that are *inputs* to the circuit. It must be non-null, but may be empty.

```
table Instance {
   header                  :Header;
   common_inputs           :[Value];
}
```

Witness Syntax. The Witness represents a *private* assignment of values to variables. It does not include variables already given in Instance nor the computable intermediary wires. The Witenss table is isomorphic to the ⟨short-witness⟩ element from section 4.2. Specifically, the Witness includes

- The header, of type Header, is included for communicating the semantics between a producer and a consumer.
- The short_witness, a vector of type Value, specify the private values assigned to wires that are *inputs* to the circuit. It must be non-null, but may be empty.

```
table Witness {
   header                  :Header;
   short_witness           :[Value];
}
```

Input Semantics (Instance & Witness). The following is a comprehensive list of the semantic checks to be performed around the instance and witness of the IR.

- *Assignment:* ensure that each Instance and Witness gate is given a `value` in the respective messages, that inputs are not defined more than once and that wires are not assigned a value more than once. It must also check that assignments are to wires in the instance or short witness wire ranges (see Section 3.2).
- *Encoding:* ensure that the `value` that each assignment is given actually encodes a field element that belongs to the underlying field, as defined by the Header. For degree 1 fields, it can be achieved by ensuring that the encoded value is strictly smaller than the field characteristic.

5.3 Relation Syntax & Semantics

The Relation resource encompasses the actual functionality of the circuit, which is represented by a set of gates. The Relation table is isomorphic to the ⟨`relation`⟩ element of section 4.4. The Relation message contains

- The `header`, of type Header, is included for communicating the semantics between a producer and a consumer.
- The `gateset`, a string, represents the concatenation of the gates supported in the circuit. The Common Gate Directives from Section 5.4.1 are assumed to be part of the circuit, so the gates to be included in this string are names the Simple Gate Directives in Section 5.4.2. If only arithmetic or boolean gates are used, one can replace the gate names by either one of the cannonical gatesets, denoted by the strings `arithmetic` and `boolean`. Partial gate sets are also allowed as a restriction of the canonical ones as a concatenated string, comma-separated, of the individual gates `@add/@addc/@mul/@mulc` and `@and/@xor/@and` as a subset of the arithmetic and boolean canonical gateset, respectively.
- The `features`, a string representing the concatenation of the different possible advanced features. The `@function`, `@for` and/or `@switch` toggles can be used to support any of the Function, GateFor and/or GateSwitch gates respectively. When no "advanced" features are supported, the features toggle expects the `@simple` string.
- The `functions`, an array of type Function, used as a declarative list of the custom functions to be used in the circuit. This array will contain all the function declarations used in the circuit. Within the circuit, only these Functions can be invoked. This must be non-null, but may be empty. If this `Relation` message is one of a sequence, the `functions` array must be empty if a prior `directives` element was non-empty.
- The `directives`, an array of type Directive, describing the actual directives that form the relation. Within a sequence of `Relation` messages, at least one message must have a non-empty `directives`.

```
table Relation {
   header                    :Header;
   gateset                   :string;
   features                  :string;
   functions                 :[Function];
   directives                :[Directive];
}
```

Relation Semantics. The following is a comprehensive list of the semantic checks to be performed around the relation resource of the IR.

- *Gateset validity:* ensure that defined gateset is either `arithmetic` (or a subset) or `boolean` (or a subset). If the gateset is `boolean` (or a subset), one must check that the field characteristic is exactly 2. This is a *resource validity* check.
- *Gate coherence:* ensure that the gates used in the relation are coherent with the `gateset`, as described above. This is a *resource validity* check.
- *Constants:* ensure that the constants used are actual field elements, for both the assignment, or `@addc`/`@mulc` gates. This is a *resource validity* check.
- *Wire consistency:* ensure that the input wires of gates map to an already existing identifier of a wire. Furthermore, one must enforce *Single Static Assignment* by checking that the same wire is used only once as an output wire. This is a *resource validity* check.
- *Assertions:* ensure that each of the `assert_zero` gates is satisfied. This is an *evaluation validity* check.
- *Consumed inputs:* Has components in both *resource* and *evalutaion validity*.
  - For *resource validity*, ensure that each function-gate (named or anonymous) consumes all the instances/witnesses it declares.
  - For *Evaluation validity:* ensure that no leftover instances/witnesses remain after processing the top level scope.

5.4 List of Directives

As explained above, each gate specifies an operation or an assignment of wires, hence defining a different fan-in and fan-out per gate. Starting in the IR v1.0.0, we differentiate between two types of gates. First, *simple gates* are gates that perform a single operation (e.g.: addition, multiplication, etc.) and are only allowed to have fan-in 2 and fan-out 1, as per the Bristol format. Starting in IR v1.0.0, the standard allows the use of *advanced gates*, which enable more complex computational structures such as loops, branches and generic functions. Advanced gates can be used as a way to reduce the size of the circuit by refactoring components that can be reused, preventing duplication.

5.4.1 Common Gate Directives

The following gates are common to all circuit types given that they specify the sources and sinks of the circuits, or the inputs to the witnessed statement. Specifically, there are six of these gates:

Constant: which output a constant value, assigned from the get go. The *constant* represents a field element encoded least significant byte first. The GateConstant table is isomorphic to the ⟨assign⟩ directive of section 4.5.

```
table GateConstant {
   output      :Wire;
   constant    :[ubyte];
}
```

Assert: which checks that the outputs assert to zero. The GateAssertZero table is isomorphic to the ⟨assert-zero⟩ directive of section 4.5.

```
table GateAssertZero {
   input       :Wire;
}
```

Copy: which copies the value of the input wire onto the output wire. In some cases this may be very useful for optimizations and other reasons. The GateCopy table is isomorphic to the ⟨copy⟩ directive of section 4.5.

```
table GateCopy {
   output      :Wire;
   input       :Wire;
}
```

Free: which frees a set of wires from memory, allowing these identifiers to be reused for other instance or witness assignments. The GateFree table is isomorphic to the ⟨delete-single⟩ and ⟨delete-range⟩ directives of section 4.5. GateFree defines a conditional structure, while ⟨delete-single⟩ and ⟨delete-range⟩ define similar structures as individual directives. The `last` attribute may be null, indicating the `first` attribute carries equivalence to ⟨delete-single⟩. If the `last` attribute is non-null, the `GateFree` carries equivalence to ⟨delete-range⟩.

```
table GateFree {
   // First wire ID to free.
   first       :Wire;
   // Last wire ID is optional.
   // Free the range [first; last] inclusive.
   last        :Wire;
}
```

Instance: which reads a value from the instance input stream. The GateInstance and GateWitness tables collectively form an isomorphism to the ⟨input⟩ directive of section 4.5. ⟨input⟩ defines conditional structure, while `GateInstance` and `GateWitness` define similar structures as individual tables.

```
table GateInstance {
   output       :Wire;
}
```

Short Witness: which reads a value from the short witness input stream.

```
table GateWitness {
   output       :Wire;
}
```

5.4.2 Simple Gate Directives

The following gates are of arithmetic nature, to be used with values in fields of a prime characteristic. These gates form the canonical `arithmetic` gate set in the Relation. In `GateAddConstant` and `GateMulConstant` there is a `constant` attribute of array type. This carries a field element, in a byte-wise decomposition. Bytes are little end first, and may have leading zero bytes, same as the `Value` table of section 5.1.

```
table GateAdd {
   output       :Wire;
   left         :Wire;
   right        :Wire;
}

table GateMul {
   output       :Wire;
   left         :Wire;
   right        :Wire;
}

table GateAddConstant {
   output       :Wire;
   input        :Wire;
   constant     :[ubyte];
}

table GateMulConstant {
   output       :Wire;
   input        :Wire;
   constant     :[ubyte];
}
```

The following gates are of boolean nature, to be used with values in fields of characteristic $= 2$. These gates form the canonical `boolean` gate set in the Relation.

```
table GateAnd {
   output       :Wire;
   left         :Wire;
   right        :Wire;
}

table GateXor {
   output       :Wire;
```

```
        left         :Wire;
        right        :Wire;
    }

    table GateNot {
        output       :Wire;
        input        :Wire;
    }
```

Together, GateAdd, GateMul, GateAnd, and GateXor form an isomorphism to ⟨binary-gate⟩ ("binary" refers to the number of inputs) of section 4.5. This is because ⟨binary-gate⟩ has conditional structure, whereas these define the conditional structure as part of their table type. Similarly GateAddConstant and GateMulConstant form an isomorphism to ⟨binary-const-gate⟩, and GateNot forms an isomorphism to ⟨unary-gate⟩.

5.4.3 Feature: Functions Gate

The function gate feature enables users to package a set of directives that can be identified by the backend as a specific function (sometimes referred to as *custom gates*, potentially one that can be computed with backend-specific optimizations. Custom gates are instantiated by different modes of operation. The first is using the Function declaration together with the GateCall invocation. This is the only way to use function gates as a stand-alone gate within a circuit (as long as the function is declared in the function array in the Relation.

The Function table represents the definition and declaration of a function gate. It is isomorphic to ⟨function-declare⟩ of section 4.6. The Function table includes

- The name, a string that allows the GateCall to invoke the function by reference. This enables uniformity of the custom gate.
- The output_count, input_count, instance_count and witness_count, all integers that allow to keep track of the number of outputs, inputs, instances and witnesses used in the function gate.
- The body, an array of directives, which describe the exact computation of the function gate. The body can also be seen as a reference implementation of the specific function. The body must be non-null and have length strictly greater than 0.

```
    table Function {
        // Declare a Function gate as a custom computation
        name              :string;
        output_count      :uint64;
        input_count       :uint64;
        instance_count    :uint64;
        witness_count     :uint64;
        body              :[Directive];
    }
```

The GateCall represents the invocation of a function gate. It is isomorphic to ⟨function-invoke⟩ of section 4.6. The GateCall table includes

- The name of the function, a string that refers to the concrete function declaration. This reduces the size of the circuit by preventing multiple copies of the same sub-circuit.

- The output_wires and input_wires, of type WireList, which are the specific wire identifiers to be used as inputs or outputs of the function gate. These identifiers are re-mapped to the internal scope to be used within the concrete directives, which hold references to generic wire identifiers.

```
table GateCall {
    // Invokes a previously defined Function gate
    name                   :string;
    output_wires           :WireList;
    input_wires            :WireList;
}
```

The AbstractGateCall is similar to the GateCall table, sharing most elements with it. It lacks the output_wires attribute, allowing it to be used within scope of either a GateFor or a GateSwitch. This table is isomorphic to ⟨case-function-invoke⟩ of section 4.8.

```
table AbstractGateCall {
    // Invokes a previously defined Function gate
    name                   :string;
    input_wires            :WireList;
}
```

The second way to use a function gate is by invoking directly the following anonymous function gate, which plays the combined role of the *declaration* and *invocation*. Anonymous function gates cannot however be used explicitly within the body of the circuit, but within the body of either a GateFor or a GateSwitch instead.

The GateAnonCall represents the invocation of an anonymous function gate. These are defined once at the place of their invocation. It is isomorphic to ⟨anon-function⟩ of section 4.6. The contents are a combination of the Function and GateCall, except without the name string, since there is no referencing needed.

```
table GateAnonCall {
    output_wires     :WireList;
    inner            :AbstractAnonCall;
}
```

Portions of the GateAnonCall are replaced with the AbstractAnonCall table, to share elements with the AbstractAnonCall which is isomorphic to ⟨case-anon-function⟩ of section 4.8. The subcircuit attribute must be non-null and have length strictly greater than 0.

```
table AbstractAnonCall {
    input_wires      :WireList;
    instance_count   :uint64;
    witness_count    :uint64;
    subcircuit       :[Directive];
}
```

Function gate semantics.

- *Name labels:* ensure that the name string in the Function declaration is in compliance with the labeling defined in Appendix A as ⟨label⟩. This is a *syntax validity* constraint.
- *Feature allowed:* ensure that the string @function is included in the feature toggle in the Relation. This is a *resource validity* constraint.
- *Wiring consistency:* ensure that the size of the input_wires and output_wires arrays in the GateCall is the same as the input_count and the output_count in the Function declaration, respectively. This is a *resource validity* constraint.
- As a*resource validity* constraint, recursion is prohibited by requiring each nested function call is either anonymous or to a prior defined function.

5.4.4 Feature: Switch-Case Statements

The switch-case statements enable circuits to represent conditional branching within the relation, in a way that enables backends to compute a single branch without revealing which one. This means that The GateSwitch table is isomorphic to the ⟨switch-statement⟩ directive of section 4.8, while CaseInvoke and CaseInvokeU collectively form an isomrphism to ⟨case-function⟩ also from section 4.8. The GateSwitch table includes

- The condition element, of type Wire, which carries the value that determines which branch is taken.
- The output_wires, of type WireList, specify the wires to be used as the output of the switch gate.
- The branch cases, an array of values used to match with the condition element and decide which branch is to be taken. This array must be non-null and have length strictly greater than zero.
- The branches, an array of CaseInvoke (as defined below), which contains the set of all possible branches and the sub-circuits that belong to that computation. This array must be non-null and have the same length as the cases attribute.

```
table GateSwitch {
    condition          :Wire;
    output_wires       :WireList;
    cases              :[Value];
    branches           :[CaseInvoke];
}
```

The following are helper tables that enable the use of a two-dimensional array of directives in the body of the for loop, by invoking either a AbstractGateCall or a AbstractAnonCall.

```
union CaseInvokeU {
    AbstractGateCall,
    AbstractAnonCall,
}

table CaseInvoke {
    invocation  :CaseInvokeU;
}
```

Switch gate semantics.

- *Condition matching:* ensure that the condition wire matches exactly one of the cases values. There is no *default* branch.
- *Feature allowed:* ensure that the string @switch is included in the feature toggle in the Relation.

5.4.5 Feature: For Loops

The GateFor provides circuits with the functionality of *for loops*, enabling the iteration over the same block of computation without repeating the block itself. The GateFor table and ForLoopBody union collectively form an isomorphism to ⟨for-loop⟩ of section 4.7. The ForLoopBody union represents conditional structure within the ⟨for-loop⟩ element. The GateFor table contains

- The outputs, a WireList that references the outputs of the for loop.
- The iterator, a string denoting the name of the iterator to be used within the iterator expressions in the ForLoopBody below. It must have the same form as ⟨label⟩.
- The range of the iterator denoted by first and last, inclusive.
- The body of the loop over which the iterations take place.

```
    table GateFor {
      outputs      :WireList;
      iterator     :string;
      first        :uint64;
      last         :uint64;
      body         :ForLoopBody;
    }
```

```
     union ForLoopBody {
       IterExprFunctionInvoke,
       IterExprAnonFunction
     }
```

We now define the *iterators* and *iterator expressions*. The IterExprWireNumber is a type used uniquely within *for loops* to allow computing expressions on values that differ per iteration of the loop. The IterExprWireNumber is equivalent to the ⟨iter-expr-wire-number⟩ of section 4.7.

```
table IterExprWireNumber {
    value       :IterExpr;
}
```

Throughout this section we refer to these as *iterators* or *iterator expressions*. The following structures form simple arithmetic expressions which dereference wires using for-loop iterators. The IterExpr union and its members form an isomorphism to the ⟨iter-expr⟩ element of section 4.7.

```
union IterExpr {
    IterExprConst,
    IterExprName,
    IterExprAdd,
    IterExprSub,
    IterExprMul,
    IterExprDivConst,
}

// Constant value in an expression
table IterExprConst { value :uint64; }
// Named loop-iteration
table IterExprName { name : string; }
// Sum of two expressions
table IterExprAdd { left :IterExprWireNumber; right :
   IterExprWireNumber; }
// Difference of two expressions
table IterExprSub { left :IterExprWireNumber; right :
   IterExprWireNumber; }
// Product of two expressions
table IterExprMul { left :IterExprWireNumber; right :
   IterExprWireNumber; }
// Division of an expression by a constant
table IterExprDivConst { numer :IterExprWireNumber; denom :uint64;
   }
```

As a named string, the iterator is referenced in the computation. This is equivalent to naming a specific identifier in memory that has a mutable value assigned to it. The allowed expressions are addition, substraction and multiplication of two iterators, whether named strings or constant values, as well as division of a (potentially) named iterator and a constant value.

In a similar fashion to the WireRange above, we provide the following structures to enable support of compact lists using iterators. The IterExprWireRange table, the IterExprWireListElementU and IterExprWireListElement tables, and the IterExprWireList table form isomorphisms to the ⟨iter-expr-wire-list-range⟩ element, the ⟨iter-expr-wire-list-element⟩ element, and the ⟨iter-expr-wire-list⟩ element, respectively, all from section 4.7. The ⟨iter-expr-wire-list-single⟩ has no FlatBuffer isomorphism, as it is simply a decoration for illustration of semantics. The elements attribute of IterExprWireList must be non-null but may be empty.

```
table IterExprWireRange {
    first :IterExprWireNumber;
    last  :IterExprWireNumber;
}
```

```
union IterExprWireListElementU {
    IterExprWireNumber,
    IterExprWireRange,
}

table IterExprWireListElement {
    element     :IterExprWireListElementU;
}

table IterExprWireList {
    elements    :[IterExprWireListElement];
}
```

The following describe the concrete `body` types allowed within the GateFor. These are equivalent to the anonymous functions described above, except that they also allow for iterator expressions, as needed for the for loops, as described in Section 3.6. The IterExprFunctionInvoke table is isomorphic to ⟨`iter-expr-function-invoke`⟩ and the IterExprAnonFunction table is isomorphic to ⟨`iter-expr-anon-function`⟩, both of section 4.7. The `body` attribute of `IterExprAnonFunction` must be non-null and non-empty.

```
table IterExprFunctionInvoke {
  name      :string;
  outputs   :IterExprWireList;
  inputs    :IterExprWireList;
}
```

```
table IterExprAnonFunction {
  outputs           :IterExprWireList;
  inputs            :IterExprWireList;
  instance_count    :uint64;
  witness_count     :uint64;
  body              :[Directive];
}
```

For loop semantics.

- *Feature allowed:* ensure that the string `@for` is included in the feature toggle in the Relation. This is a *resource validity* check.
- *Iterator scoping:* ensure that the loop iterator must be in lexical scope as part of the resource validity.
- *Output consistency:* ensure that each iterations outputs are distinct, and the union of all iteration's outputs are the same as the loops outputs.

Directives List

All of the directives that can be used as gates in the circuit are referenced as part of a union of gates called the FlatBuffers DirectiveSet. We differentiate this directive set from the string type in the Relation in that this one is specific to FlatBuffers and defines all the possible gates that can be used in circuits, as part of the schema; whereas the gateset in the Relation is specific to a circuit instance and is used for the semantics of the circuit itself.

The DirectiveSet union is isomorphic to ⟨directive⟩ of section 4.4 when combined with the table Directive from 5.1. There is no direct isomorphism to the ⟨directive-list⟩ from 4.4, however an array, [Directive], takes its place at each occurrence.

```
union DirectiveSet {
   GateConstant,
   GateAssertZero,
   GateCopy,
   GateAdd,
   GateMul,
   GateAddConstant,
   GateMulConstant,
   GateAnd,
   GateXor,
   GateNot,
   GateInstance,
   GateWitness,
   GateFree,
   GateCall,
   GateAnonCall,
   GateSwitch,
   GateFor,
}
```

6 Suggested Feature Reductions

The SIEVE IR allows three (3) sub-scope features which may be disabled during statement generation. Although a backend may prefer that certain features be disabled, this may not always be the case. This section outlines procedures and guidelines for a backend to reduce an unwanted feature to wanted features. The procedures given here assume that the circuit is semantically valid as described in section 4, and does not make any effort to warn about invalidities.

As a reminder, when any of the sub-scope features are enabled, the high order wire numbers (2^{63} through $2^{64} - 1$) are reserved as “ephemeral” wires – for the backend to use. This section makes heavy use of these “ephemeral” wires.

6.1 Inlining Function Gates

The reduction for a function-gate is to “inline” it, or copy its body into the body of the caller. Where a function gate calls other function gates, it may be most optimal to work from the inner-most function gate to the outer most. Here is the recommended procedure for inlining a single function gate.

1. Copy the body of the function gate into the body of the caller.

2. For each wire mentioned in the copied body:

   (a) If the wire is one of the function gate's outputs or inputs, replace it with the corresponding wire from the calling scope.

   (b) If the wire is local to the function gate, replace its assignment with an ephemeral wire. Then track the association between local and ephemeral for replacing the wires uses as gate inputs.

3. When the end of the function gate's body is reached, each ephemeral wire should be deleted.

6.2 Unrolling For Loops

Since a for loop's number of repetitions may be calculated statically, its body may be "unrolled", or repeated the correct number of times. Since for loops may be nested, this procedure must be recurse to remove the loop iterator from iterator expressions within its body.

1. The number of repetitions may be calculated as `last - first + 1`.

2. The for loop should be replaced by number of repetitions many invocations of the loop's body (the body is defined to be a function gate invocation).

3. The input and output lists of each function gate contain iterator expressions, instead of wire numbers. Each of these expressions must be evaluated against this loop's, and any containing loops', iterator(s).

4. If the body of the loop is an anonymous function gate, then the unrolling process must recurse into the function gate's body, passing its own iterator and its containing loop's iterators into the function gate. Named function gate invocations do not require checking, because they do not have access to containing loop's iterators.

5. within the recursed body, the bodies of anonymous function and of switch-case statements must be checked recursively.

Once the unrolling procedure is complete, loop body functions may be inlined using the procedure described in section 6.1.

6.3 Multiplexing Switch-Case Statements

Flattening a switch-case statement should be a simple matter of multiplexing the outputs of of each case. With the *Boolean gate set* this is a simple matter. For the *arithmetic gate set* a multiplexer is described here. Complications arise due to directives with side-effects, namely `@assert_zero`, `@instance`, and `@short_witness`.

1. Using the switch wire's value, w, create a selector bit, s_c, for each case, c, $s_c = 1 - (w - c)^{p-1} \mod p$. This works because of Fermat's Little Theorem.

2. Replace each case's output wires with ephemeral wires.
3. Multiply each case's ephemeral wires by s_c.
4. Sum the output wires across each case to produce the final output wires.
5. Assert that the sum of all s_c is exactly 1, otherwise either no case was selected, or a duplicated case was selected.

The @assert_zero directive must be disabled in non-selected cases. This should be achievable by replacing each one with a function-gate output, and multiplexing these "assert outputs" from each case into a single @assert_zero directive. This suggested procedure may be applied recursively from the bottom up to handle nested switch-cases.

1. If the case's function is named and used elsewhere, duplicate it, along with all the named functions which it invokes, if they are used elsewhere.
2. Count the number of @assert_zero directives within the case, including those in function-invocations and for-loops (if this procedure is applied from the bottom up, there should be no sub-switch-case statements).
3. Add that many additional outputs to the case using ephemeral wires.
4. Within the case's body, as well as functions invocations and for loops within the body, replace @assert_zero directives with copy directives to the additional outputs.
5. Once all cases are completed, multiplex the maximum number of additional outputs, into @assert_zero directives, omitting values from the multiplexer where one case has fewer additional wires than another.

The @instance and @short_witness directives consume values from streams. The semantics of a switch-case statement demand that it consumes as many values from each stream as the maximum consumption of its cases. In order to flatten this behavior, we consume all stream values ahead of the cases, and pass them in as input wires. The following procedure should be applied once for each of the @instance and @short_witness.

1. Find the maximum consumption.
2. Consume this many values from the stream as ephemeral wires.
3. If necessary duplicated each case's body, and replace its stream usage count with additional input wires.
4. Replace each usage of the stream directive with a copy directive.
5. Renumber local wires to account for additional input wires.

As a late addition to this specification, the following scenarios may cause the automated translation to multiplexers to fail. Both scenarios involve nesting for-loops within switch-statements.

1. If a loop traverses a range of wires which starts in the output or input range and continues into the input or local range, the automated insertion additional inputs or outputs will cause disruptions in loop's traversal range.
2. It is possible to construct a useful loop with varying number of output-wires on each iteration, so long as the loop has no input-wires. However when converting instances/witneses to input-wires this loop is no-longer valid.

In both of these scenarios, the correct behavior would be maintained by unrolling the loop before multiplexing a surrounding switch.

7 Contributors

Authors

- Team Wizkit: Paul Bunn, David Darais, Daniel Genkin, Steve Lu, Kimberlee Model, Tarik Riviere, Muthuramakrishnan Venkitasubramaniam, Xiao Wang
- Team Emphasize: Steven Eker, Karim Eldefrawy, Stéphane Graham-Lengrand, Vitor Pereira, Hadas Zeilberger
- Team Fromager: Michael Adjedj, Daniel Benarroch Guenun, Eran Tromer, Aurélien Nicolas, Constance Beguier
- Team Oracles: Mayank Varia

The following individuals from the T&E Team were instrumental in moderating the IR's development: David A. Wilson (PI), R. Nicholas Cunningham, J. Parker Diamond, Hanson Duan, Ariel Hamlin, Noah Luther, Richard Shay.

Disclaimers

This material is based upon work supported by DARPA under Contracts No. HR001120C0087, HR001120C0086, HR001120C0085 and Agreement No. HR00112020021. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

Distribution Statement A: Approved for public release. Distribution Unlimited.

The authors of the SIEVE IR have made every effort that the SIEVE IR Specification, if adopted by a recognized standards body, would be compatible with the OSI Open Standard Requirement.

No Intentional Secrets We have made every effort to ensure that the IR contains no necessary secrets. Further, several of the SIEVE Performers have or will open-source their IR implementations.

Availability The SIEVE Performers have agreed to release the text of the IR Specification under the Creative Commons Attribution 4.0 International (CC-BY 4.0) terms.

Patents Although we are not aware of any current patents which may be necessary to implement the IR, the SIEVE Performers have agreed not to enforce any intellectual property rights which may be necessary to implement the IR.

Dependencies To our knowledge, all technologies dependent by the IR are already themselves open-source.

Appendix A Textual Syntax Building Blocks

This Appendix describes the building blocks used by section 4 of this specification. It will briefly overview the number formats and identifiers used by the SIEVE IR. Throughout this section both tokens (terminal symbols) and nonterminal symbols are described, thus indications of the intended interpretation are given.

The IR defines binary, octal, decimal, and hexadecimal literals as interchangeably, with conventional prefixes `0b`, `0o`, and `0x` for binary, octal, and hexadecimal literals. Decimal literals have no prefixes, but must not have leading zeros. All numeric literals represent unbounded positive integers. A complete attribute grammar is provided later in this appendix.

```
⟨numeric-literal$_0$⟩ ::= ⟨binary-literal$_1$⟩ | ⟨octal-literal$_1$⟩
                      | ⟨decimal-literal$_1$⟩ | ⟨hex-literal$_1$⟩
  ⟨numeric-literal$_0$⟩.value := match {
    case ⟨binary-literal$_1$⟩: ⟨binary-literal$_1$⟩.value;
    ...
    case ⟨hex-literal$_1$⟩: ⟨hex-literal$_1$⟩.value;
  }
```

The IR uses numeric identifiers for wires, or ⟨wire-number⟩. These carry an index into the *wirset* described by section 4.3.1. Although ⟨wire-number⟩s are written here in BNF, they are to be interpreted as tokens, not allowing space between elements.

```
⟨wire-number$_0$⟩ ::= ‘$’ ⟨numeric-literal$_1$⟩
  ⟨wire-number$_0$⟩.index := ⟨numeric-literal$_1$⟩.value;
```

The IR defines the following syntax for a Field Element Literal. The Field Element in general is an element of the Galois Field defined for the statement. For $GF(p^1)$, this is a numeric literal $\mathrm{mod}\ p$ encased with angle braces. The ⟨field-literal⟩ should be interpreted as allowing spaces between elements.

```
⟨field-literal$_0$⟩ ::= ‘<’ ⟨numeric-literal$_1$⟩ ‘>’
  fail_if(⟨numeric-literal$_1$⟩.value >= $p$);
  ⟨field-literal$_0$⟩.value := ⟨numeric-literal$_1$⟩.value;
```

For referencing previously defined function gates and loop iterators, the ⟨label⟩ is used as a lookup key. Its form is intended to be similar to a C-style identifier, but allows “grouping” similar names with separators ‘.’ and ‘::’. It should be interpreted as a token, disallowing spaces.

```
⟨label⟩ ::= [‘a’-‘z’ | ‘A’-‘Z’ | ‘_’]  [‘a’-‘z’ | ‘A’-‘Z’ | ‘0’-‘9’ | ‘_’]*
        (
           [‘.’ | ‘::’]
           [‘a’-‘z’ | ‘A’-‘Z’ | ‘_’]  [‘a’-‘z’ | ‘A’-‘Z’ | ‘0’-‘9’ | ‘_’]*
        )*
```

The full format for numeric literals is given here. They take the form of positive, unbounded integers in one of binary, octal, decimal, or hexadecimal format. To indicate binary, octal, and hexadecimal, a `0b`, `0o`, or `0x` prefix is used. Decimal numbers have no prefix, but may not have leading zeros, to avoid confusion with C-Style octal numerals (leading zero, with no "o" distinguisher). Although numeric literals are written here in BNF, they are to be interpreted as tokens, not allowing space between elements.

```
⟨binary-literal$_0$⟩ ::= [ ‘0b’ | ‘0B’ ] [ ⟨binary-digit$_{1..m}$⟩ ]+
  ⟨binary-literal$_0$⟩.value := 0;
  for(i from 1 to m) {
    ⟨binary-literal$_0$⟩.value *= 2;
    ⟨binary-literal$_0$⟩.value += ⟨binary-digit$_i$⟩.value;
  }
⟨binary-digit$_0$⟩ ::= ‘0’ | ‘1’
  ⟨binary-digit$_0$⟩.value := match {
    case ‘0’: 0;
    case ‘1’: 1;
  }

⟨octal-literal$_0$⟩ ::= ‘0o’ [ ⟨octal-digit$_{1..m}$⟩ ]+
  ⟨octal-literal$_0$⟩.value := 0;
  for(i from 1 to m) {
    ⟨octal-literal$_0$⟩.value *= 8;
    ⟨octal-literal$_0$⟩.value += ⟨octal-digit$_i$⟩.value;
  }
⟨octal-digit$_0$⟩ ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’
  ⟨octal-digit$_0$⟩.value := match {
    case ‘0’: 0;
    ...
    case ‘7’: 7;
  }


⟨decimal-literal$_0$⟩ ::= ‘0’
  ⟨decimal-literal$_0$⟩.value := 0;
⟨decimal-literal$_0$⟩ ::= ⟨no-zero-decimal-digit$_1$⟩ [ ⟨decimal-digit$_{1..m}$⟩ ]*
  ⟨decimal-literal$_0$⟩.value := ⟨no-zero-decimal-digit$_1$⟩;
  for(i from 1 to m) {
    ⟨decimal-literal$_0$⟩.value *= 10;
    ⟨decimal-literal$_0$⟩.value += ⟨decimal-digit$_i$⟩.value;
  }
⟨no-zero-decimal-digit$_0$⟩ ::= ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ |
                             ‘7’ | ‘8’ | ‘9’
  ⟨no-zero-decimal-digit$_0$⟩.value := match {
    case ‘1’: 1;
    ...
    case ‘9’: 9;
  }
⟨decimal-digit$_0$⟩ ::= ‘0’ | ⟨no-zero-decimal-digit$_1$⟩
  ⟨decimal-digit$_0$⟩ := match {
    case ‘0’: 0;
    case ⟨no-zero-decimal-digit$_1$⟩: ⟨no-zero-decimal-digit$_1$⟩.value;
  }

⟨hex-literal$_0$⟩ ::= [ ‘0x’ | ‘0X’ ] [ ⟨hex-digit$_{1..m}$⟩ ]+
  ⟨hex-literal$_0$⟩.value := 0;
  for(i from 1 to m) {
    ⟨hex-literal$_0$⟩.value *= 16;
    ⟨hex-literal$_0$⟩.value += ⟨hex-digit$_i$⟩.value;
  }
⟨hex-digit$_0$⟩ ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’
              | ‘8’ | ‘9’ | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’
              | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’
  ⟨hex-digit$_0$⟩.value := match {
    case ‘0’: 0;
    ...
    case ‘9’: 9;
    case ‘a’: 10;
    ...
    case ‘f’: 15;
    case ‘A’: 10;
    ...
    case ‘F’: 15;
  }
```

The IR text format will also allow “comments” of the following form, which will be treated as whitespace during parsing.

```
  ⟨comment⟩ ::= ‘//’ [ ‘{any character except \n}’ ]* ‘\n’
  ⟨comment⟩ ::= ‘/*’ [ ‘{any character except the sequence ‘*/’}’ ]* ‘*/’
```