The P4 Language Specification

Version 1.0.2

March 3, 2015

The P4 Language Consortium

1 Introduction

P4 is a declarative language for expressing how packets are processed by the pipeline of a network forwarding element such as a switch, NIC, router or network function appliance. It is based upon an abstract forwarding model consisting of a parser and a set of match+action table resources, divided between ingress and egress. The parser identifies the headers present in each incoming packet. Each match+action table performs a lookup on a subset of header fields and applies the actions corresponding to the first match within each table. Figure 1 shows this model.

P4 itself is protocol independent but allows for the expression of forwarding plane protocols. A P4 program specifies the following for each forwarding element.

- *Header definitions:* the format (the set of fields and their sizes) of each header within a packet.
- *Parse graph:* the permitted header sequences within packets.
- *Table definitions:* the type of lookup to perform, the input fields to use, the actions that may be applied, and the dimensions of each table.
- *Action definitions:* compound actions composed from a set of primitive actions.
- *Pipeline layout and control flow:* the layout of tables within the pipeline and the packet flow through the pipeline.

P4 addresses the configuration of a forwarding element. Once configured, tables may be populated and packet processing takes place. These post-configuration operations are referred to as "run time" in this document. This does not preclude updating a forwarding element's configuration while it is running.

1.1 The P4 Abstract Model

The following diagram shows a high level representation of the P4 abstract model.

The P4 machine operates with only a few simple rules.

© 2014-2015, The P4 Language Consortium

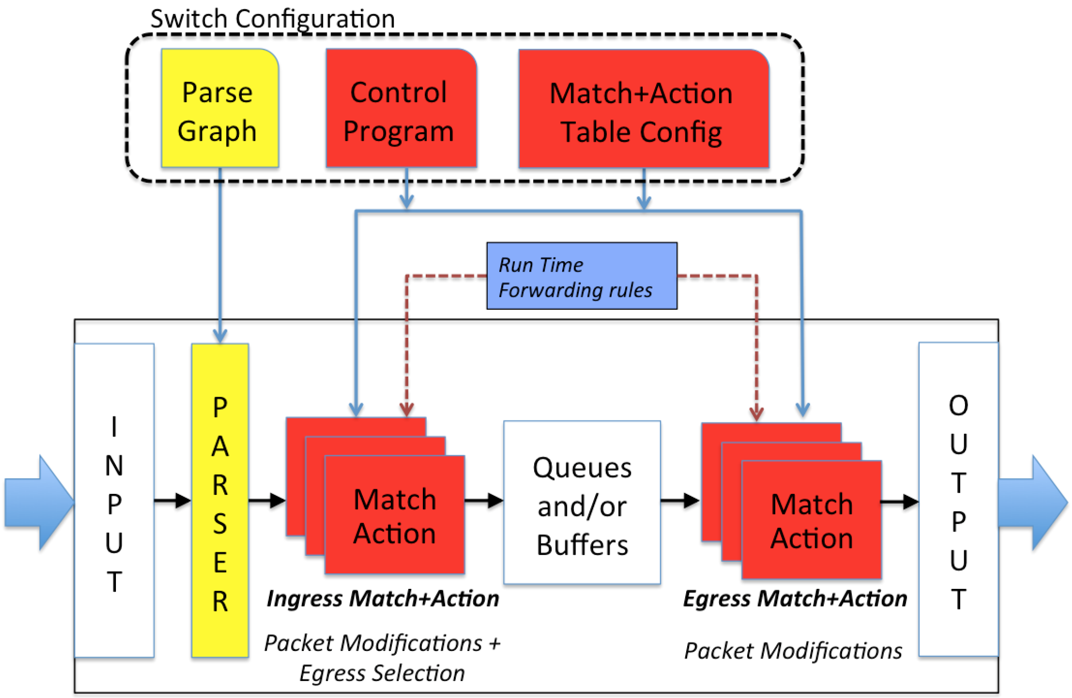

Figure 1: Abstract Forwarding Model

- For each packet, the parser produces a *Parsed Representation* on which match+action tables operate.
- The match+action tables in the *Ingress Pipeline* generate an *Egress Specification* which determines the set of ports (and number of packet instances for each port) to which the packet will be sent.
- The *Queuing Mechanism* processes this Egress Specification, generates the necessary instances of the packet and submits each to the *Egress Pipeline*. Egress queuing may buffer packets when there is over-subscription for an output port, although this is not mandated by P4.
- A packet instance's physical destination is determined before entering the *Egress Pipeline*. Once it is in the Egress Pipeline, this destination is assumed not to change (though the packet may be dropped or its headers further modified).
- After all processing by the Egress Pipeline is complete, the packet instance's header is formed from the Parsed Representation (as modified by match+action processing) and the resulting packet is transmitted.

Although not shown in this diagram, P4 supports recirculation and cloning of packets. This is described in detail in Section 14.

P4 focuses on the specification of the parser, match+action tables and the control flow through the pipelines. Programmers control this by writing a P4 program which specifies the switch configuration as shown at the top of Figure 1.

A machine that can run a P4 program is called *target*. Although a target may directly execute a P4 program, it is assumed in this document that the program is compiled into a suitable configuration for the target.

In the current version, P4 does not expose, for example, the functionality of the Queuing Mechanism and does not specify the semantics of the Egress Specification beyond what is mentioned above. Currently they are defined in target specific input to the compiler and exposed in conjunction with other interfaces that provide run time system management and configuration. Future versions of P4 may expose configuration of these mechanisms allowing consistent management of such resources from the P4 program.

1.2 The mTag Example

The original P4 paper [1] includes an example called mTag. We use this example throughout this specification as a means of explaining the basic language features as they are presented. Complete source for this example, including sample run-time APIs, is available at the P4 web site, [2].

We give an overview of the mTag example here. Quoting from the original paper:

> Consider an example L2 network deployment with top-of-rack (ToR) switches at the edge connected by a two-tier core. We will assume the number of end-hosts is growing and the core L2 tables are overflowing. . . . P4 lets us express a custom solution with minimal changes to the network architecture. . . . The routes through the core are encoded by a 32-bit tag composed of four single-byte fields. The 32-bit tag can carry a "source route".... Each core switch need only examine one byte of the tag and switch on that information. [1]

Two P4 programs are defined for this example: One for edge switches (called "ToR" above) and one for aggregation switches (called "core switches" above). These two programs share definitions for packet headers, the parser and actions.

1.3 P4 Abstractions

P4 provides the following abstractions. A P4 program consists of instances of each.

- *Header type:* A specification of fields within a header.
- *Header instance:* A specific instance of a packet header or metadata.

- *Parser state function:* Defines how headers are identified within a packet.
- *Action function:* A composition of primitive actions that are to be applied together.
- *Table instance:* Specified by the fields to match and the permitted actions.
- *Control flow function:* Imperative description of the table application order.
- *Stateful memories:* Counters, meters and registers which persist across packets.

In addition to these high level abstractions, the following are used

- For a header instance:
  - *Metadata:* Per-packet state which may not be derived from packet data. Otherwise treated the same as a packet header.
  - *Header stack:* a contiguous array of header instances.
  - *Dependent fields:* Fields whose values depend on a calculation applied to other fields or constants.
- For a parser:
  - *Value set:* run-time updatable values used to determine parse state transitions.
  - *Checksum calculations:* The ability to apply a function to a set of bytes from the packet and test that a field matches the calculation.

1.4 Structure of the P4 Language

This section (work in progress) provides a brief overview of the structure of the P4 language.

The P4 language uses a flat typing structure, inferring types for most function parameters. In general each P4 level declaration has its own namespace, though potential ambiguities are identified in the spec.

Constant values can be expressed in P4 in binary, decimal and hexadecimal. Base specifications 0x and 0b are used to indicate binary and hexadecimal respectively.

It is sometimes necessary to indicate the number of bits that should be used to represent the value. P4 allows this by means of a width indication preceding the base specification. See Section 1.5.1 below.

P4 allows value expressions with operators so long as they can be evaluated at compile time.

1.5 Specification Conventions

This document represents P4 grammatical constructs using BNF with the following conventions:

- The BNF is presented in green boxes.
- Terminal nodes are indicated with **bold**.
- A node with a name ending in `_name` is implicitly a string whose first character is a letter (not a digit).
- Nodes followed by + indicate one or more instances.
- Nodes followed by `*` indicate zero or more instances.
- A vertical bar, `|`, separates options from which exactly one must be selected.
- Square brackets, `[]`, are used to group nodes. A group is optional unless it is followed by +. A group may be followed by * indicating zero or more instances of the group.
- Symbols with special significance (e.g., `[ ] * + |`) may be used as terminal nodes by enclosing them in quotes: for example `"*"`.
- Symbols other than those listed above are literals. Examples include curly braces, colon, semi-colon, parentheses, and comma.
- If a rule does not fit on one line, a new line immediately follows `::=` and the description ends with a blank line.
- Examples code appears in blue boxes to differentiate them more clearly from BNF rules.
- To emphasize those locations where a field width is indicated (the width of the value's representation may matter), the node `field_value` is used. This is a synonym for any other constant value, `const_value`.

Header types and table definitions are specified declaratively. These typically consist of a set of attribute/value pairs separated by a colon.

Parsers, actions and control flow are specified imperatively with untyped parameters (if any) and a limited set of operations.

1.5.1 Value Specifications

As noted above in Section 1.4, P4 supports generic and bit-width specific values. These are unified through the following representation.

```
const_value ::= [ "+" | - ] [ width_spec ] unsigned_value
unsigned_value ::= binary_value | decimal_value | hexadecimal_value

binary_value ::=  binary_base binary_digit+
decimal_value ::= decimal_digit+
hexadecimal_value ::= hexadecimal_base hexadecimal_digit+

binary_base ::= 0b | 0B
hexadecimal_base ::= 0x | 0X

binary_digit ::= _ | 0 | 1
decimal_digit ::= binary_digit | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hexadecimal_digit ::=
    decimal_digit | a | A | b | B | c | C | d | D | e | E | f | F

width_spec ::= decimal_digit+ '
field_value ::= const_value
```

The width specification is followed by an apostrophe (not a back-tick). The width must be specified in decimal.

Note that constants always start with a digit to distinguish them from other identifiers.

The node `const_value` may be read as "constant value". The node `field_value` is used in this specification to emphasize that the width of the representation may be relevant; otherwise it is a synonym for `const_value`.

Whitespace terminates a constant specification.

Underscores are permitted in values to add clarity by grouping digits; they are ignored otherwise. Examples include: `78_256_803` (replacing commas in decimal representation) or `0b1101_1110_0101` (grouping bits into nibbles or bytes in binary representation).

Optionally, the bit-width of the value may be specified as indicated by `bit_width`. Zero is permitted as a width which is the same as not specifying a width. If no width precedes the value, then the width is inferred. For positive values the inferred width is the smallest number of bits required to contain the value. For negative values the inferred width is one more than the smallest number of bits required to contain the positive value.

Negative numbers are represented in two's complement. See Appendix 17.5. Field Value Conversions regarding conversions and sign extension of field values.

Here are some example values.

Notation	Decimal Value	Bit Width	Notes
42	42	6	Default base is decimal
16'42	42	16	The same value, but explicitly given a width of 16 bits.
0b101010	42	6	Binary representation of same with implicit width
0'0x2a	42	6	A zero width is the same as not specifying a width meaning the width is inferred from the value.
12'0x100	256	12	Example of bit width and hexadecimal base indication.
7'0b1	1	7	Binary value specified with explicit width
-0B101	-5	4	The negative is not applied until the rest of the value is evaluated.

Table 1: Value Representation Examples

Expressions, including the use of parentheses, are supported for both arithmetic and field values with the expectation that they are evaluated by the compiler. The following binary integer operators are allowed with their syntax, semantics,

precedence and associativity matching that of C:

```
+ - * / % « » | & ˆ ~
```

2 Headers and Fields

2.1 Header Type Declarations

Header types describe the layout of fields and provide names for referencing information. Header types are used to declare header and metadata instances. These are discussed in the next section.

Header types are specified declaratively according to the following BNF:

```
header_type_declaration ::=
   header_type header_type_name { header_dec_body }

header_dec_body ::=
    fields { field_dec + }
```

```
    [ length : length_exp ; ]
    [ max_length : const_value ; ]

field_dec ::= field_name : bit_width [ ( field_mod ) ];
field_mod ::= signed | saturating | field_mod , field_mod
length_bin_op ::= "+" | - | "*" | "<<" | ">>"
length_exp ::=
    const_value |
    field_name |
    length_exp length_bin_op length_exp |
    ( length_exp )

bit_width ::= const_value | "*"
```

Header types are defined with the following conventions.

- Header types must have a `fields` attribute.
  - The list of individual fields is ordered.
  - Fields are, by default, unsigned and non-saturating (i.e., addition/subtraction causing overflow/underflow will wrap).
  - The bit offset of a field from the start of the header is determined by the sum of the widths of the fields preceding it in the list.
  - Bytes are ordered sequentially (from the packet ordering).
  - Bits are ordered within bytes by most-significant-bit first. Thus, if the first field listed in a header has a bit width of 1, it is the high order bit of the first byte in that header.
  - All bits in the header must be allocated to some field.
  - One field at most within a header type may specify a width of `"*"` which indicates it is of variable length.
- If all fields are fixed width (no fields of width `"*"`) then the header is said to be of *fixed length*. Otherwise it is of *variable length*.
- The `length` attribute specifies an expression whose evaluation gives the length of the header in bytes for variable length headers.
  - It must be present if the header has variable length (some field has width `"*"`).
  - A compiler warning must be generated if it is present for a fixed length header.

  - Fields referenced in the length attribute must be located before the variable length field.
- The `max_length` attribute indicates the maximum allowed length of the header in bytes for a variable length header.
  - If, at run time, the calculated length exceeds this value, it is considered a parser exception. See Section 4.6.
  - The `max_length` attribute may be present if the header is variable length.
  - A compiler warning must be generated if it is present for a fixed length header.
- Operator precedence and associativity follows C programming conventions.

P4 supports variable-length headers for packet headers via the use of fields with the special bit width `"*"`. The width of such a field is inferred from the total header length (which is in bytes) as indicated by the `length` attribute. The width of the field in bits is `((8 * length) - sum-of-fixed-width-fields)`. Only one field at most within a header may specify a width of `"*"`.

An example declaration for a VLAN header (802.1Q) is:

```
header_type vlan_t {
    fields {
        pcp             : 3;
        cfi             : 1;
        vid             : 12;
        ethertype       : 16;
    }
}
```

Metadata header types are declared with the same syntax.

```
header_type local_metadata_t {
    fields {
        cpu_code        : 16; // Code for packet going to CPU
        port_type       : 4;  // Type of port: up, down, local...
        ingress_error   : 1;  // An error in ingress port check
        was_mtagged     : 1;  // Track if pkt was mtagged on ingr
        copy_to_cpu     : 1;  // Special code resulting in copy to CPU
        bad_packet      : 1;  // Other error indication
    }
}
```

2.2 Header and Metadata Instances

While a header type declaration defines a header *type*, a packet may contain multiple instances of a given type. P4 requires each header instance to be declared explicitly prior to being referenced.

There are two sorts of header instances: packet headers and metadata. Usually, packet headers are identified from the packet as it arrives at ingress while metadata holds information about the packet that is not normally represented by the packet data such as ingress port or a time stamp.

Most metadata is simply per-packet state used like registers while processing a packet. However, some metadata may have special significance to the operation of the switch. For example, the queuing system may interpret the value of a particular metadata field when choosing a queue for a packet. P4 acknowledges these target specific semantics, but does not attempt to represent them.

Packet headers (declared with the `header` keyword) and metadata (declared with the `metadata` keyword) differ only in their validity. Packet headers maintain a separate valid indication which may be tested explicitly. Metadata is always considered to be valid. This is further explained in Section 2.2.1. Metadata instances are initialized to 0 by default, but initial values may be specified in their declaration.

The BNF for header and metadata instances is:

```
instance_declaration ::= header_instance | metadata_instance
header_instance ::= scalar_instance | array_instance
scalar_instance ::= header header_type_name instance_name ;
array_instance ::=
    header header_type_name
        instance_name "[" const_value "]" ;

metadata_instance ::=
   metadata header_type_name
       instance_name [ metadata_initializer ] | ;

metadata_initializer ::= { [ field_name : field_value ; ] + }
```

Some notes:

- Only packet headers (not metadata instances) may be arrays (header stacks).
- `header_type_name` must be the name of a declared header type.
- Metadata instances may not be declared with variable length header types.

- The fields named in the initializer must be from the header type's `fields` list.
- If an initializer is present, the named fields are initialized to the indicated values; unspecified values are initialized to 0.
- For header instances, the compiler must produce an error if the total length of all fields in a header type is not an integral number of bytes. The compiler may pad the header to be byte aligned.

For example:

```
header vlan_t inner_vlan_tag;
```

This indicates that space should be allocated in the Parsed Representation of the packet for a `vlan_t` header. It may be referenced during parsing and match+action by the name `inner_vlan_tag`.

A metadata example is:

```
metadata local_metadata_t local_metadata;
```

This indicates that an `local_metadata_t` type object called `local_metadata` should be allocated for reference during match+action.

2.2.1 Testing if Header and Metadata Instances are Valid

Packet headers and their fields may be checked for being *valid* (that is, having a defined value). Validity and deparsing (see Section 5) are the only points where packet headers and metadata headers differ.

A header instance, declared with the keyword `header`, is *valid* if it is extracted during parsing (see 5. Parser Specification, below) or if an action makes it valid (add or copy). A field (inside a header instance) is valid if its parent header instance is valid.

All fields in a metadata instance are always valid. Testing a metadata field for validity should raise a compiler warning and will always evaluate to True.

> **Explanation:** The reason for this is best seen by examining the case of a "flag"; for example, suppose a one bit metadata flag is used to indicate that a packet has some attribute (say, is an IP packet, v4 or v6). There is no practical difference between the flag having a value of 0 and the flag itself being invalid. Similarly, many "index" metadata fields can be given a reserved value to indicate they are invalid (hence support for initial values of metadata fields). While occasionally it would be useful to have an independent valid bit for a metadata

field, defining a separate metadata flag to represent that field's validity is a reasonable work around.

Only valid packet header fields may result in a match (when a value is specified for exact or ternary matches against the field), although a match operation may explicitly check if a header instance (or field) is valid. Only valid packet headers are considered for deparsing (see Section 5).

2.2.2 Header Stacks

P4 supports the notion of a *header stack* which is a sequence of adjacent headers of the same type. MPLS and VLAN tags are examples that might be treated this way. Header stacks are declared as arrays as shown in the `array_instance` clause in Section 2.2.

Header stack instances are referenced using bracket notation and such references are equivalent to a non-stack instance reference. The parser maintains information to manage the header stack. See the proposal in Appendix 17.8.2. Parser Repeat Loops regarding how they may be parsed.

2.3 Header and Field References

For match, action and control flow specifications, we need to make references to header instances and their fields. Headers are referenced via their instance names. For header stacks, an index is specified in square brackets.

```
header_ref ::= instance_name | instance_name "[" index "]"
index ::= const_value | last
```

To refer to a particular header field, we use dotted notation. The keyword `last` can be used as an index to refer to the largest-index valid instance of a header stack.

```
field_ref ::= header_ref . field_name
```

For example `inner_vlan_tag.vid` where `inner_vlan_tag` has been declared as an instance of header type `vlan_tag`.

- Field names must be listed in the `fields` attribute of the header declaration.
- A field reference is always relative to its parent header. This allows the same field name to be used in different header types without ambiguity.
- Each header instance may be valid or invalid at any given time. This state may be tested in match+action processing.

- References at run time to a header instance (or one of its fields) which is not valid results in a special "undefined" value. The implications of this depend on the context.

2.4 Field Lists

In many cases, it is convenient to specify a sequence of fields. For example, a hash function may take a sequence of fields as input or a checksum may be calculated based on a sequence of fields. P4 allows such declarations. Each entry may be a specific field instance reference, a header instance (which is equivalent to listing all the header's fields in order) or a fixed value. Packet headers and metadata may be referenced in a field list.

```
field_list_declaration ::=
    field_list field_list_name {
        [ field_list_entry ; ] +
    }

field_list_entry ::=
    field_ref | header_ref | field_value | field_list_name | payload
```

A field list may reference other field lists. As a result, field lists names and header instance names should be considered part of the same namespace. Recursive references are not supported.

The identifier `payload` indicates that the contents of the packet following the header of the previously mentioned field is included in the field list. This is to support special cases like the calculation of an Ethernet CRC across the entire packet or the TCP checksum.

3 Checksums and Hash-value generators

Checksums and hash value generators are examples of functions that operate on a stream of bytes from a packet to produce an integer. These have many applications in networking. The integer may be used, for example, as an integrity check for a packet or as a means to generate a pseudo-random value in a given range on a packet-by-packet or flow-by-flow basis.

P4 provides a means of associating a function with a set of fields and allowing the resulting operation (a map from packets to integers) to be referenced in P4 programs. These are called field list calculations or calculation objects. P4 does not support the expression of the algorithm for computing the underlying function, treating these like primitive actions. A set of known algorithms are identified for convenience.

The resulting functions – a field list calculation maps a packet to an integer – may be configurable through run time APIs. Targets may vary in their support of these interfaces, but typically the seed value of the calculation may be configured, the algorithm may have configurable parameters (such as the coefficients for a polynomial used in the calculation) and possibly even the set of fields used may be configured.

The field list may be referenced as a field property for checksums, discussed in Section 3.1, or referenced in a primitive action.

```
field_list_calculation_declaration ::=
    field_list_calculation field_list_calculation_name {
        input {
            [ field_list_name ; ] +
        }
        algorithm : stream_function_algorithm_name ;
        output_width : const_value ;
    }
```

Run time APIs allow the selection of one of the input field lists to be active at a time. The first listed name is used as the default.

The `output_width` value is in bits.

A field instance is excluded from the calculation (i.e., it is treated as if the instance is not listed in the input list) if the field's header is not valid.

The algorithm is specified as a string. The following algorithms are defined with the given names, and targets may support others.

- **xor16:** Simply the XOR of bytes taken two at a time.
- **csum16:** See the IPv4 header checksum description in https://tools.ietf.org/html/rfc791#page-14.
- **crc16:** See http://en.wikipedia.org/wiki/Crc16.
- **crc32:** See http://en.wikipedia.org/wiki/Crc32
- **programmable_crc:** This algorithm allows the specification of an arbitrary CRC polynomial. See http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

3.1 Checksums

Some fields, such as the IP checksum, hold the result of a stream calculation. P4 allows the representation of these dependencies with the calculated field declaration. Calculated fields matter to the extent they are verified at packet ingress or are updated at packet egress.

The syntax associates a sequence of update or verify directives to a specific field instance, each of which may have a condition associated with it. The first entry with a condition satisfied by the packet (or with no condition specified) determines the association. This complexity allows the selection of different calculations based on the packet's format. For example, the calculation of a TCP checksum may vary slightly based on whether the packet has an IPv4 or an IPv6 header.

Note that the conditions are evaluated at the point the verify or update operations are carried out.

Currently only limited conditions are supported.

```
calculated_field_declaration ::=
    calculated_field field_ref { update_verify_spec + }

update_verify_spec ::=
    update_or_verify field_list_calculation_name [ if_cond ] ;

update_or_verify ::= update | verify
if_cond ::= if ( calc_bool_cond )
calc_bool_cond ::=
    valid ( header_ref | field_ref ) |
    field_ref == field_value
```

Here is an example declaration. It assumes `field_list_calculation` declarations for `tcpv4_calc` and `tcpv6_calc` have been given and that `ipv4` and `ipv6` are packet header instances.

```
calculated_field tcp.chksum {
        update tcpv4_calc if (valid(ipv4));
        update tcpv6_calc if (valid(ipv6));
        verify tcpv4_calc if (valid(ipv4));
        verify tcpv6_calc if (valid(ipv6));
}
```

For checksums, the field list calculation is intended to bind the field list and algorithm to a specific field instance. This declaration indicates that the value stored in `field_ref` is expected to be the value calculated by the given field set calculation on the packet. Note that although this declaration may occur anywhere in the P4 program, the declaration **should** be placed immediately after the header instance declaration for the field referenced.

Fields that are variable length (width is "*") are not allowed to be declared as calculated fields.

The `verify` option indicates that the parser should calculate the expected value and check if that value is stored in the indicated field. If the value is not equal, then a `p4_pe_checksum` exception is generated; see 6.6.1. Standard Parser Exceptions. This check occurs at the end of parsing and is performed only if `field_ref` is valid.

The `update` option indicates that the system should update the value of the field if changes are made to any fields on which it depends. The update to the field occurs when the packet is deparsed for egress. If no update clause applies, the field retains its value from the match+action pipeline.

4 Parser Specification

P4 models the parser as a state machine. This can be represented as a parse graph with each state a node and the state transitions as edges. Figure 2 shows a very simple example. Note that this figure identifies a header with each state. While P4 supports this approach, it does not require it. A node in the parse graph may be purely a decision node and not bound to a particular header instance, or a node may process multiple headers at once.

Here are a few of the P4 parser functions for the mTag parser. The start function calls ethernet directly.

```
parser ethernet {
    extract(ethernet);   // Start with the ethernet header
    return select(latest.ethertype) {
        0x8100:     vlan;
        0x800:      ipv4;
        default:    ingress;
    }
}

parser vlan {
    extract(vlan);
    return select(latest.ethertype) {
        0xaaaa:     mtag;
```

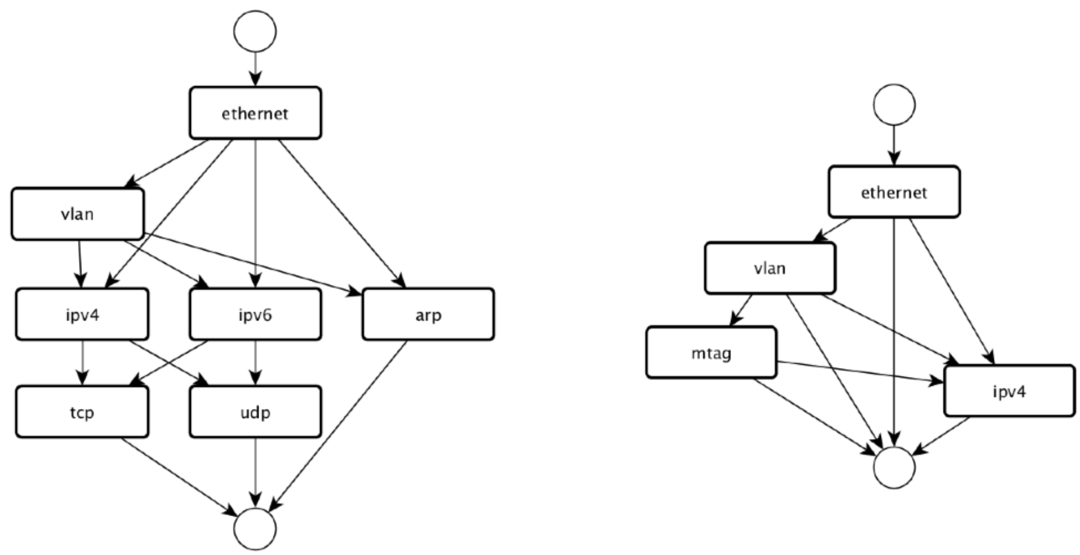

Figure 2: Simple Parse Graph and mTag Parse Graph

```
        0x800:      ipv4;
        default:    ingress;
    }
}

parser mtag {
    extract(mtag);
    return select(latest.ethertype) {
        0x800:      ipv4;
        default:    ingress;
    }
}
```

The reference to `ingress` terminates parsing and invokes the ingress control flow function.

4.1 Parsed Representation

The parser produces the representation of the packet on which match+action stages operate. This is called the *Parsed Representation* of the packet. It is the set of header instances which are valid for the packet. The parser produces the initial Parsed Representation as described below. Match+action may update the Parsed Representation of the packet by modifying field values and by changing which header instances are valid; the latter results in adding and removing headers.

The Parsed Representation holds packet headers as they are updated by match+action. The original packet data may be maintained for special operations such as cloning, described in Section 14.

Metadata is considered part of the Parsed Representation for the packet as it is generally treated like other packet headers.

4.2 Parser Operation

The parser is fed the packet from the first byte. It maintains a *current offset* into the packet which is a pointer to a specific byte in the header. It extracts headers from the packet at the current offset into per-packet header instances and marks those instances valid, updating the Parsed Representation of the packet. The parser then moves the current offset forward (indicating the next valid byte of the packet to process) and makes a state transition.

The P4 program may examine metadata in making state transition decisions, though targets may have limitations on this ability. For example, the ingress port may be used to determine an initial parser state allowing of different packet formats. Similarly, the metadata provided by cloning or recirculation can be used to change the parsing behavior for such packets; see Section 14.

In P4, each state is represented as a parser function. A parser function may exit in one of four ways:

- A return statement specifying the name of a parser function is executed. This parser function is the next state to which the machine must transition.
- A return statement specifying the name of a control function (as described in Section 12) is executed. This terminates parsing and begins match-action processing by calling the indicated control function.
- An explicit `parse_error` statement executes. See Section 4.6 for more information.
- An implicit parser error occurs. These are described in Section 4.6.1.

Note that because of the first two points, parser function names and control function names share a common namespace. The compiler must raise an error if two such functions have the same name.

A `select` operation is defined to allow branching to different states depending on expressions involving fields or packet data.

If headers are to be extracted when entering a state, these are signaled explicitly by calls to an extract function (defined in 7.5. The extract Function) at the beginning of the parser function definition (defined in 7. Parser Specification).

4.3 Value Sets

In some cases, the values that determine the transition from one parser state to another need to be determined at run time. MPLS is one example where the value of the MPLS label field is used to determine what headers follow the MPLS tag and this mapping may change dynamically at run time. To support this functionality, P4 supports the notion of a Parser Value Set. This is a named set of values with a run time API to add and remove values from the set. The set name may be referenced in parse state transition conditions (the value list in a case entry).

Parser Value Sets contain values only, no header types or state transition information. All values in a value set must correspond to the same transition. For example, all MPLS labels corresponding to an IPv4 transition would exist in one set, while all MPLS labels corresponding to an IPv6 transition would exist in a different set.

Value sets are declared at the top level of a P4 program, outside of parser functions. There is a single global namespace for value sets. They should be declared before being referenced in parser functions.

```
value_set_declaration ::= parser_value_set value_set_name;
```

The width of the values is inferred from the place where the value set is referenced. If the set is used in multiple places and they would infer different widths, then the compiler must raise an error.

The run time API for updating parser value sets must allow value and mask pairs to be specified together.

4.4 Parser Function BNF

Here is the BNF for declaring a parser function:

```
parser_function_declaration ::=
    parser parser_state_name { parser_function_body }

parser_function_body ::=
    extract_or_set_statement*
    return_statement

extract_or_set_statement ::= extract_statement | set_statement
```

```
extract_statement ::= extract ( header_extract_ref );
header_extract_ref ::=
     instance_name |
     instance_name "[" header_extract_index "]"

header_extract_index ::= const_value | next

set_statement ::= set_metadata ( field_ref, metadata_expr ) ;
metadata_expr ::= field_value | field_or_data_ref

return_statement ::=
    return_value_type |
    return select ( select_exp ) { case_entry + }

return_value_type ::=
    return parser_state_name ; |
    return control_function_name ; |
    parse_error parser_exception_name ;

case_entry ::= value_list : case_return_value_type ;
value_list ::= value_or_masked [ , value_or_masked ]* | default

case_return_value_type ::=
    parser_state_name |
    control_function_name |
    parse_error parser_exception_name

value_or_masked ::=
    field_value | field_value mask field_value | value_set_name

select_exp ::= field_or_data_ref [, field_or_data_ref] *
field_or_data_ref ::=
    field_ref |
    latest.field_name |
    current( const_value , const_value )
```

The extract function can only extract to packet headers, not to metadata.

Select functions take a comma-separated list of fields and concatenate their values, with the left-most field forming the most-significant bits of the concatenated value. The select operation then compares the values in the order they occur in the program to the entries to find a matching one.

The `mask` operator is used to indicate a ternary match should be performed using the indicated mask value. The comparison between the select expression and the case's value is limited to the bits set in the mask; that is, the select expression and value are each ANDed with the mask before the comparison is made.

Allowing masked matches and value sets means that more than one of the cases could match. The order of cases determines which takes precedence: the first case in the list that matches is used.

The header reference `latest` refers to the most recently extracted header instance within the parse function. It is an error to reference `latest` without a preceding `extract` operation in the same function.

The field reference `current(...)` allows the parser to reference bits that have not yet been parsed into fields. Its first argument is the bit offset from the current offset and its second argument is the bit width. The result is treated as an unsigned field-value of the given bit width. It is converted to the metadata field according to the conversion rules described in Appendix 17.5. Field Value Conversions.

In a `set_metadata` statement, the field referenced must be in a metadata instance. If the value has a different width than the destination metadata field, then conversion occurs as described in Section 15.7.

4.5 The extract Function

The extract function takes a header instance as a parameter. The header instance cannot be metadata. Extract copies data from the packet at the current offset into that header instance and moves the current parsing location to the end of that header.

Note that we use the special identifier **`next`** (rather than **`last`**) for header stacks as we are extracting into the next available free location.

4.6 Parser Exceptions

There are two possible treatments for errors that occur during parsing: drop or process. In the drop case, the packet may be immediately dropped by the parser. No match+action processing is done on the packet. An implementation should provide one or more counters for such events.

For the alternative, process, the parsing operation is halted, special metadata is set to indicate that a parser error occurred and the packet is passed to a control function for match+action processing. The packet is processed according to the installed match+action rules like any other packet, but those rules may check for a parser error and apply policies such as forwarding the packet to the control plane.

There are a number of error conditions recognized by P4 which may be triggered implicitly. These are listed in the table below. In addition, the programmer may signal errors with the `parse_error` exception in a parser function. They are both handled in the same manner.

Parser exception handlers may be explicitly declared by the programmer as follows. Multiple metadata set calls may be invoked followed by a directive either to return to a control function or to drop the packet. Note that setting metadata will only have an effect if return is executed.

```
parser_exception_declaration ::=
    parser_exception parser_exception_name {
        set_statement *
        return_or_drop ;
    }

return_or_drop ::= return_to_control | parser_drop
return_to_control ::= return control_function_name
```

4.6.1 Standard Parser Exceptions

A set of standard exception names are defined as follows. The prefix "pe" stands for parser exception.

Identifier	**Exception Event**
`p4_pe_index_out_of_bounds`	A header stack array index exceeded the declared bound.
`p4_pe_out_of_packet`	There were not enough bytes in the packet to complete an extraction operation.
`p4_pe_header_too_long`	A calculated header length exceeded the declared maximum value.
`p4_pe_header_too_short`	A calculated header length was less than the minimum length of the fixed length portion of the header.
`p4_pe_unhandled_select`	A select statement had no default specified but the expression value was not in the case list.
`p4_pe_checksum`	A checksum error was detected.
`p4_pe_default`	This is not an exception itself, but allows the programmer to define a handler to specify the default behavior if no handler for the condition exists.

Table 2: Standard Parser Exceptions

When an exception passes the packet for match+action processing, the exception type is indicated as metadata; see Section 6.

4.6.2 Default Exception Handling

If a handler for `p4_pe_default` is defined and an exception occurs for which no `parser_exception` handler was defined by the programmer, the `p4_pe_default` handler is invoked.

If an exception occurs, no `parser_exception` handler was defined for that exception, and no `p4_pe_default` handler is defined, then the packet is dropped by the parser.

5 Deparsing

At egress, the forwarding element converts the Parsed Representation (as updated by match+action) to a serial stream of bytes for transmission. This process is called deparsing as it reverses the process of parsing.

P4 takes the approach that any format which should be generated on egress should be represented by the parser used on ingress. Thus, the parse graph represented in the P4 program is used to determine the algorithm used to produce the serialized packet from the Parsed Representation. Note the following considerations:

- Only headers which are valid are serialized.
- If the parse graph is acyclic, then a topological ordering (that is, a linear order that respects the parse graph's ordering) can be generated and used to determine the order by which headers should be serialized.
- In general, cycles occur in the parse graph when parsing header stacks or a set of optional headers. These may be treated as a single node in the parse graph and serialized as a group.
- Metadata fields are not serialized directly (as they are not parsed). Metadata fields may be copied to packet header fields in match+action processing, allowing them to be serialized for egress.

6 Standard Intrinsic Metadata

Metadata is state associated with each packet. It can be treated like a set of variables associated with each packet, read and written by actions executed by tables. However, some metadata has special significance to the operation of the switch. This is called *Intrinsic Metadata* as it has semantics intrinsic to the operation of the machine. Examples include the ingress port number or the egress selection. The first is an example of read only data which is set by the switch when the packet arrives; the second is set by table actions, but then is processed by the Buffer Mechanism and results in the packet being sent to a particular egress port or ports.

This specification identifies *Standard Intrinsic Metadata* fields for which support is mandatory for P4 compliant targets. Although these fields are mandatory, the format of these fields may be target specific. The definition for these formats must be provided by the target, either as a header to be automatically included by a compiler, or internally in the compiler's implementation.

Standard Intrinisic Metadata is called out in this section either because it is automatically populated (`ingress_port` for instance) or because it is necessary to describe how the abstract machine operates (`egress_port` for instance).

Targets may provide their own definitions of intrinsic metadata, although programs which depend on such definitions may not be portable.

This table shows the fields defined for the metadata instance `standard_metadata`:

Field	Notes
`ingress_port`	The port on which the packet arrived. Set prior to parsing. Always defined. Read only.
`packet_length`	The number of bytes in the packet. For Ethernet, does not include the CRC. Set prior to parsing. Cannot be used for matching or referenced in actions if the switch is in "cut-through" mode. Read only.
`egress_spec`	Specification of an egress. Undefined until set by match+action during ingress processing. This is the "intended" egress as opposed to the committed physical port(s) (see `egress_port` below). May be a physical port, a logical interface (such as a tunnel, a LAG, a route, or a VLAN flood group) or a multicast group.
`egress_port`	The physical port to which this packet instance is committed. Read only. This value is determined by the Buffering Mechanism and so is valid only for egress match+action stages. See Section 13 below. Read only.
`egress_instance`	An opaque identifier differentiating instances of a replicated packet. Read only. Like `egress_port`, this value is determined by the Buffering Mechanism and is valid only for egress match+action stages. See Section 13 below.
`instance_type`	Represents the type of instance of the packet: • normal • ingress clone • egress clone • recirculated TBD: Either flags or type value; for example, could a clone be recirculated? Do we need a counter too?
`parser_status`	Result of the parser. 0 means no error. Otherwise, the value indicates what error occurred during parsing. Specific representation is TBD.
`parser_error_location`	If a parser error occurred, this is an indication of the location in the parser program where the error occurred. Specific representation is TBD.

Table 3: Standard Intrinsic Metadata Fields

7 Counters, Meters and Registers

Counters, meters and registers maintain state for longer than one packet. Together they are called stateful memories. They require resources on the target and hence are managed by a compiler.

In this section, we refer to an individual counter, meter or register as a *cell*. In P4, stateful memories are organized into named arrays of cells (all of the same type of object). A cell is referenced by its array name and index. Cells are accessed or updated by the actions applied by a table. Targets may have limitations on the amount of computation that can be done to determine the index of the cell being accessed. They may also have limitations on the updates that can be done to the cell's contents.

For example:

```
counter ip_pkts_by_dest {
    type : packets;
    direct : ip_host_table;
}
```

declares a set of counters attached to the table named ip_host_table. It allocates one counter cell for each entry in that table.

Another example:

```
meter customer_meters {
    type : bytes;
    instance_count : 1000;
}
```

declares an array of 1000 meters named customer_meters. These may be referenced from the actions of any table (though usually only one or two tables will be likely to reference them).

P4 allows stateful memory resources to be global – that is, referenced by any table – or to be static – bound to one table instance. Normally, multiple table entries, whether or not they are in the same table, may refer to the same cell. This is called *indirect access*. P4 also allows *direct access* where the stateful memory resource is bound to one table and each entry in the table is allocated its own dedicated cell in that memory. An example of this is where every table entry has its own counter.

A compiler will attempt to allocate the resources required by the program according to availability on the target. However, target constraints may make this impossible; for example, a target may not allow references to the same global resource in both the ingress and egress pipelines.

Counters and meters are referenced in special primitive actions as defined in Section 9.1. Registers may be used as arguments to the same primitive actions that modify header fields.

7.1 Counters

Counters are declared as follows.

```
counter_declaration ::=
    counter counter_name {
        type : counter_type ;
        [ direct_or_static ; ]
        [ instance_count : const_value ; ]
        [ min_width : const_value ; ]
        [ saturating ; ]
    }

counter_type ::= bytes | packets
direct_or_static ::= direct_attibute | static_attribute
direct_attribute ::= direct : table_name
static_attribute ::= static : table_name
```

The `min_width` attribute indicates the minimum number of bits required for each cell. The compiler or target may allocate more bits to each cell.

The `saturating` attribute indicates that the counter will stop counting if it reaches its maximum value (based on its actual bit-width). Otherwise the counter will wrap.

If the counter is declared with the `direct` attribute, one counter is associated with each entry in the named table. In this case, no count action needs to be given for the table actions; they are automatically updated whenever the corresponding entry is applied. As a result, counter names declared as direct are not allowed to be referenced in the `count` primitive and a compiler must raise an error if this occurs.

Run time APIs should be provided to indicate the actual width of a given counter. This is necessary for calculating the maximum value a counter may take (which is necessary for properly managing saturation or roll over).

If the counter is not declared `direct`, actions must reference the counter by name and index.

If the counter is declared with the `static` attribute, the counter resource is dedicated to the indicated table. The compiler must raise an error if the counter name is referenced by actions used in another table.

The instance_count attribute indicates the number of instances (cells) of the counter to allocate. The instance_count attribute is **required** if the counter is not declared with the direct attribute. The compiler should raise an error if both instance_count and direct are specified together, or if neither direct nor instance_count are specified.

7.2 Meters

Meter declarations follow those of counters.

```
meter_declaration ::=
    meter meter_name {
        type : meter_type ;
        result : field_ref ;
        [ direct_or_static ; ]
        [ instance_count : const_value ; ]
    }

meter_type ::= bytes | packets
```

Meters may be declared to measure packets or bytes. The configuration of meters is not defined by P4, but a meter is assumed to return a status (usually called a color). That status is stored in the field specified by the result attribute.

If the meter is declared with the direct attribute, one meter is associated with each entry in the named table. In this case, no meter index is required to determine which cell is being used. However the meter call is needed to identify where the return state is stored.

If a meter is declared with the static attribute, it may only be referenced by actions invoked in the indicated table. The compiler must raise an error if a different table attempts to invoke an action with this meter.

The instance_count attribute indicates the number of instances (cells) of the meter to allocate. The instance_count attribute is **required** if the meter is not declared with the direct attribute.

7.3 Registers

Registers are stateful memories whose values can be read and written in actions. They are like counters, but can be used in a more general way to keep state.

A simple example use might be to verify that a "first packet" was seen for a particular type of flow. A register cell would be allocated to the flow, initialized to "clear". When the protocol signalled a "first packet", the table would match on this value and update the flow's cell to "marked". Subsequent packets in the flow could would be mapped to the same cell; the current cell value would be stored in metadata for the packet and a subsequent table could check that the flow was marked as active.

Register declarations are similar to those of meters and counters. Registers may be declared either with a width or with a header type layout.

```
register_declaration ::=
    register register_name {
        width_or_layout ;
        [ direct_or_static ; ]
        [ instance_count : const_value ; ]
        [ attribute_list ; ]
    }

width_or_layout ::= width_declaration | layout_declaration
width_declaration ::= width : const_value ;
layout_declaration ::= layout : header_type_name ;

attribute_list ::= attributes : attr_entry
attr_entry ::= signed | saturating | attr_entry , attr_entry
```

Field names must be listed in the `fields` attribute of the header declaration.

The `instance_count` attribute indicates the number of instances (cells) of the register to allocate. The `instance_count` attribute is **required** if the register is not declared with the `direct` attribute.

Although registers cannot be used directly in matching, they may be used as the source of a `modify_field` action allowing the current value of the register to be copied to a packet's metadata and be available for matching in subsequent tables.

If a register is declared with a layout declaration, the header type must be fixed length (no "*" width fields).

A register reference is done with array syntax.

```
register_reference ::=
    register_name "[" const_value "]" [.field_name]
```

If the register is declared with a layout, then the reference can be refined with a field name as indicated.

8 Match+Action Table Overview

P4 allows the specification of table instances with the `table` declaration. This declaration defines the exact set of fields that should be examined to find a match (a "hit"). Associated with each entry is an indication of an action to take should the entry match.

If no entry is found that matches the current packet, the table is said to "miss"; in this case a default action for the table may be applied.

Each entry in a match+action table has the following parts:

- The match values for comparison with the Parsed Representation of the packet. The format of these values determined by the table declaration.
- A reference to an action function, if the entry should match. The set of allowed action functions is specified in the table declaration.
- Parameter values to pass to the action when the action function is called. The format of these parameters is determined by the particular action function selected by the entry.

9 Actions

In P4, actions are declared imperatively as functions. These function names are used when populating the table at run time to select the action associated with each entry. These are called *compound actions* to differentiate them from *primitive actions*, or simply *actions* when the context is clear.

Action functions take parameters. The values passed to these parameters are programmed into the table entry by the run time API. When that entry is selected due to a match, those parameters are passed to the action. The P4 table declarations might be used to generate run time APIs which would have parameters corresponding to the action parameters for the entry's action. Typically, the compiler would be responsible for ensuring that the values in the run time APIs are properly mapped to and consistent with the P4 program specification.

In addition to values from the matching table entry, the action operation has access to headers and metadata in the Parsed Representation.

Action functions are built from primitive actions. A standard set of primitive actions are listed in the following section, although a target may support additional target-specific primitives. Using target-specific primitives limits the portability of the resulting program.

Here are two example functions from the mTag example. The first indicates a copy of the packet should be sent to the CPU. The parameters cpu_code and bad_packet are exposed to the run time API and will be set according to the values provided when a table entry is added.

```
// Copy the packet to the CPU;
action common_copy_pkt_to_cpu(cpu_code, bad_packet) {
    modify_field(local_metadata.copy_to_cpu, 1);
    modify_field(local_metadata.cpu_code, cpu_code);
    modify_field(local_metadata.bad_packet, bad_packet);
}
```

This function sets up the mTag. It would only be invoked on a edge switch.

```
// Add an mTag to the packet; select egress spec based on up1
action add_mTag(up1, up2, down1, down2) {
    add_header(mtag);
    // Copy VLAN ethertype to mTag
    modify_field(mtag.ethertype, vlan.ethertype);

    // Set VLAN's ethertype to signal mTag
    modify_field(vlan.ethertype, 0xaaaa);

    // Add the tag source routing information
    modify_field(mtag.up1, up1);
    modify_field(mtag.up2, up2);
    modify_field(mtag.down1, down1);
    modify_field(mtag.down2, down2);

    // Set the destination egress port as well from the tag info
    modify_field(standard_metadata.egress_spec, up1);
}
```

9.1 Primitive Actions

P4 supports an extensible set of primitive actions. Primitive actions can be declared as part of a P4 program. This allows the front-end of a compiler to verify calls to primitive actions. Only the parameter list is identified for primitive actions; no body is defined.

```
primitive_action_declaration ::=
    primitive_action action_name "(" [ param_list ] ")" ;
```

```
param_list ::= param_name [, param_name]*
```

Not all targets will support all actions. Target switches may have limits on when variables are bound and what combinations of parameter types are allowed.

Here is a brief summary of primitive actions. More detailed documentation is below.

API name	**Summary**
`add_header`	Add a header to the packet's Parsed Representation
`copy_header`	Copy one header instance to another.
`remove_header`	Mark a header instance as invalid.
`modify_field`	Set the value of a field in the packet's Parsed Representation.
`add_to_field`	Add a value to a field.
`add`	Add two values together and store in a field.
`set_field_to_hash_index`	Apply a field list calculation and use the result to generate an offset value.
`truncate`	Truncate the packet on egress.
`drop`	Drop a packet (in the egress pipeline).
`no_op`	Placeholder action with no effect.
`push`	Push all header instances in an array down and add a new header at the top.
`pop`	Pop header instances from the top of an array, moving all subsequent array elements up.
`count`	Update a counter.
`meter`	Execute a meter operation.
`generate_digest`	Generate a packet digest and send to a receiver.
`resubmit`	Resubmit the original packet to the parser with metadata.
`recirculate`	Resubmit the packet after all egress modifications.
`clone_ingress_pkt_to_ingress`	Send a copy of the original packet to the parser. Alias: `clone_i2i`.
`clone_egress_pkt_to_ingress`	Send a copy of the egress packet to the parser. Alias: `clone_e2i`.
`clone_ingress_pkt_to_egress`	Send a copy of the original packet to the Buffer Mechanism. Alias: `clone_i2e`.
`clone_egress_pkt_to_egress`	Send a copy of the egress packet to the Buffer Mechanism. Alias: `clone_e2e`.

Table 4: Primitive Actions

Action parameters are typed as follows:

Notation	Type Description
HDR	The literal name of a header instance.
ARR	The name of a header instance array, with no subscript.
FLD	A field reference of form header_instance.field_name which refers to the Parsed Representation.
FLDLIST	A field list instance declared with field_list.
VAL	An immediate value or a value from a table entry's action parameters. The latter is represented as a parameter from the enclosing function (see examples below).
C-REF	The name of a counter array; determined at compile time.
M-REF	The name of a meter array; determined at compile time.
R-REF	The name of a register array; determined at compile time.
FLC-REF	Field list calculation reference; determined at compile time.

Table 5: Action Parameter Types

Here is the API specification for standard primitive actions.

`add_header(header_instance)`

Summary

Add a header to the packet's Parsed Representation

Parameters

`header_instance` (HDR) The name of the header instance to add.

Description

If the `header_instance` is not an element in a header stack, the indicated header instance is set valid. If the header instance was invalid, all its fields are initialized to 0. If the header instance is already valid, it is not changed.

If `header_instance` is an element in a header stack, the effect is to push a new header into the stack at the indicated location. Any existing valid instances from the given index or higher are copied to the next higher index. The given instance is set to valid. If the array is fully populated when this operation is executed, then no change is made to the Parsed Representation.

`copy_header(destination, source)`

Summary

Copy one header instance to another.

Parameters

`destination`	(HDR) The name of the destination header instance.
`source`	(HDR) The name of the source header instance.

Description

Copy all the field values from the source header instance into the destination header instance. If the source header instance was invalid, the destination header instance becomes invalid; otherwise the destination will be valid after the operation. The source and destination instances must be of the same type.

`remove_header(header_instance)`

Summary

Mark a header instance as invalid.

Parameters

`header_instance`	(HDR) The name of the header instance to remove.

Description

If the `header_instance` is not an element in a header stack, then the indicated header instance is marked invalid. It will not be available for matching in subsequent match+action stages. The header will not be serialized on egress. All field values in the header instance become uninitialized.

If the `header_instance` is an element in a header stack, the effect is to pop the indicated element from the stack. Any valid instances in the stack at higher indices are copied to the next lower index.

`modify_field(dest, value [, mask] )`

Summary

Set the value of the given field in packet's Parsed Representation

Parameters

`dest`	(FLD or R-REF) The name of the field instance to modify (destination).
`value`	(VAL, FLD or R-REF) The value to use (source).
`mask`	(VAL) An optional mask to use identifying the bits to change.

Description

Update the indicated field's value. The `value` parameter may be any of:

- An immediate value (a number).
- A value from the matching entry's action parameter data; in this case, the name of a parameter from the enclosing function is used.
- A Parsed Representation field reference.
- A register reference.

This allows the programmer to copy one field to another. If the width of the source field, `value`, or the `mask` is greater than that of the destination field, then the value in the source field is first truncated to the low order bits to fit into the destination field. If the width of the source field is less than that of the destination, it will be coerced to the larger field size according to its signedness.

If the parent header instance of `dest` is not valid, the action has no effect. If `value` is a field reference and its parent header is not valid, the operation has no effect.

If `mask` is specified, then the field becomes `(current_value & ~ mask) | (value & mask)`. If `mask` is not specified, the operation has the effect of a "set", modifying all bits of the destination.

`add_to_field(dest, value)`

Summary

Add a value to a field.

Parameters

`dest`	(FLD or R-REF) The name of the field instance to be modified.
`value`	(VAL, FLD or R-REF) The value to use.

Description

The `dest` field's value is updated by adding the value parameter. The `value` parameter may be from a table parameter, an immediate a value, a field reference or a register reference; see `modify_field` above. If `value` is a field reference and its parent header is not valid, then no change is made to `dest`. A description of the logical behavior follows in the Section 9.1.1 below. If `value` is an immediate value, it may be negative.

`add(dest, value1, value2)`

Summary

Add value1 and value2 and store in dest.

Parameters

`dest`	(FLD or R-REF) The name of the field instance to be modified.
`value1`	(VAL, FLD or R-REF) The first value to use.
`value2`	(VAL, FLD or R-REF) The second value to use.

Description

The `dest` field's value is updated with the result of adding the two value parameters. Each `value` parameter may be from a table parameter, an immediate a value, a field reference or a register reference; see `modify_field` above. If either `value` is a field reference and its parent header is not valid, then no change is made to `dest`. A description of the logical behavior follows in the Section 9.1.1 below. If a `value` is an immediate value, it may be negative.

`set_field_to_hash_index(dest, field_list_calc, base, size)`

Summary

Add a value to a field.

Parameters

`dest`	(FLD or R-REF) The name of the field instance to be modified (destination)
`field_list_calc`	(FLC-REF) The field list calculation to use to generate the hash value.
`base`	(VAL) The base value to use for the index.
`size`	(VAL) The maximum value to use for the index if > 0.

Description

The field list calculation is executed to generate a hash value. If size is not zero, the hash value is used to generate a value between `base` and `(base + size - 1)` by calculating `(base + (hash_value % size))`.
If size is 0 then the value used is `(base + hash_value)`.
Normal value conversion takes place when setting `dest` to the result.

`truncate(length)`

Summary

Truncate the packet on egress.

Parameters

`length`	(VAL) The number of bytes to transmit.

Description

Indicate that the packet should be truncated on egress. The number of bytes to transmit from the packet is indicated in the parameter to the action. If the packet has fewer bytes than `length`, then it will not be changed.
Normally this action would be specified on the egress pipeline, though this is not required.

`drop()`

Summary

Drop the packet on egress.

Description

Indicate that the packet should not be transmitted. This primitive is intended for the egress pipeline where it is the only way to indicate that the packet should not be transmitted. On the ingress pipeline, this primitive is equivalent to setting the `egress_spec` metadata to a drop value (specific to the target).

If executed on the ingress pipeline, the packet will continue through the end of the pipeline. A subsequent table may change the value of `egress_spec` which will override the drop action. The action cannot be overridden in the egress pipeline.

`no_op()`

Summary

Take no action.

Description

This indicates that no action should be taken on the packet. Control flow continues as per the current control function specification.

`push(array [, count])`

Summary

Push all header instances in an array down and add a new header at the top.

Parameters

`array`	(ARR) The name of the instance array to be modified.
`count`	(VAL) An optional value indicating the number of elements to push, by default 1.

Description

This primitive is used to make room for a new element in an array of header instances without knowing in advance how many elements are already valid. An element at index N will be moved to index N+1, and the element at index 0 will be zeroed out and set valid.

If a count is specified, elements will be shifted by count instead of 1 and count header instances will be zeroed and set valid.

This primitive leaves the array's size constant; if an array is already full, elements pushed to indices beyond the static array size will be lost.

`pop(array [, count])`

Summary

Pop header instances from the top of an array, moving all subsequent array elements up.

Parameters

`array`	(ARR) The name of the instance array to be modified.
`count`	(VAL) An optional value indicating the number of elements to pop, by default 1.

Description

This primitive is used to remove elements from an array of header instances without knowing in advance how many elements are already valid. An element at index N will be moved to index N-1, and the element at index 0 will be lost. The bottom-most elements that had nothing shifted into them are invalidated.

If a count is specified, elements will be shifted by count instead of 1.

Popping from an empty array (or popping more elements than are in the array) results in an empty array.

`count(counter_ref, index)`

Summary

Update a counter.

Parameters

`counter_ref`	(C-REF) The name of the counter array.
`index`	(VAL) The offset in the array to get a counter reference.

Description

The given counter is incremented by 1, if it is a packet counter, or by the packet length, if it is a byte counter. The counter array is determined at compile time. The index may be a table entry parameter or determined at compile time. It is an error to reference a direct-mapped counter array from this action.

`meter(meter_ref, index, field)`

Summary

Execute a meter operation.

Parameters

`meter_ref`	(M-REF) The name of the meter array.
`index`	(VAL) The offset in the array to get a meter reference. Applicable only if the meter type is indirect.
`field`	(FLD) A field reference to store the meter state.

Description

The given meter, determined by `meter_ref` and `index`, is executed. If the meter is direct, then `index` is ignored as the table entry determines which cell to reference. The length of the packet is passed to the meter. The state of meter is updated and the meter returns information (a "color") which is stored in `field`. If the parent header of `field` is not valid, the meter state is updated, but the state of the meter is discarded.

`generate_digest(receiver, field_list)`

Summary

Generate a digest of a packet and send to a receiver.

Parameters

`receiver`	(VAL) An opaque value identifying the receiver.
`field_list`	(FLDLIST) A list of field references.

Description

The indicated field list is populated with the packet's data and sent by a target-specific mechanism to an agent capable of processing the object. The specification of receivers is outside of the scope of P4. Example receivers might be the CPU through a channel parallel to that for transferring packets, or a co-processor connected by a bus dedicated to this operation.

This function might also be used to represent a self-updating operation such as address learning.

`resubmit( [ field_list ] )`

Summary

Applied in the ingress pipeline, mark the packet to be resubmitted to the parser.

Parameters

`field_list`	(FLDLIST) An optional list of metadata field references.

Description

Only valid on the ingress pipeline.

The packet is marked for resubmission. It will complete the ingress pipeline to generate any necessary metadata values. Then, the **original** packet data will be resubmitted to the parser with values of the fields in `field_list` from the ingress processing on the packet. These values replace the normal initial values of the metadata fields indicated in the initializer of the instance declaration.

If multiple resubmit actions get executed on one packet, the union of all the fields in the field lists should be resubmitted with the packet.

See Section 14 for more details.

`recirculate( [ field_list ] )`

Summary

On egress, mark the packet to be resubmitted to the parser.

Parameters

`field_list`	(FLDLIST) An optional list of metadata field references.

Description

Only valid on the egress pipeline.

The packet is marked for resubmission. It will complete the egress pipeline and be deparsed. This version of the packet is then resubmitted to the parser with values of the fields in `field_list` from the processing on the packet. These values replace the normal initial values of the metadata fields indicated in the initializer of the instance declaration.

See Section 14 for more details.

`clone_ingress_pkt_to_ingress(clone_spec, [ field_list ] )`

Summary

Generate a copy of the original packet and submit it to the ingress parser.

Parameters

`clone_spec`	(VAL) An opaque identifier indicating additional run time characteristics of the clone operation.
`field_list`	(FLDLIST) An optional list of metadata field references.

Description

This action indicates that the switch should generate a copy of the original packet (prior to any modifications from match+action) and submit it to the parser as an independent packet instance. This may occur immediately when the action executes or be deferred until the the original packet is buffered.

The original packet continues to be processed as though the clone had not been produced.

The `clone_spec` is used to allow the configuration of other target specific characteristics of the clone operation. It may be a simple identifier indicating a session. For instance, the clone operation may support truncating the cloned instance. The truncation length would be a property of the session. The concept of session is optional and the parameter may be ignored on some targets.

The cloned instance will have `instance_type` set to indicate that it is an ingress clone.

The fields indicated in `field_list` are copied to the Parsed Representation of the clone instance. These values replace the normal initial values of the metadata fields indicated in the initializer of the instance declaration (which occurs before parsing).

The function may also be referred to as `clone_i2i`.

See the Section 14 for more details.

`clone_egress_pkt_to_ingress(clone_spec [ , field_list])`

Summary

Generate a duplicate of the egress packet and submit it to the parser.

Parameters

`clone_spec`	(VAL) An opaque identifier indicating additional run time characteristics of the clone operation.
`field_list`	(FLDLIST) An optional list of metadata field references.

Description

The packet is marked for cloning at egress. Once the original packet completes the egress pipeline, a copy of the deparsed packet (including all modifications due to match+action) is passed to the parser as an independent packet instance. The original packet is forwarded as normal.

The `clone_spec` is used to allow the configuration of other target specific characteristics of the clone operation as described in `clone_ingress_pkt_to_ingress`.

The fields indicated in `field_list` are copied to the clone instance. These values replace the normal initial values of the metadata fields indicated in the initializer of the instance declaration.

The cloned instance will have `instance_type` set to indicate that it is an ingress clone.

The function may also be referred to as `clone_e2i`.

See the Section 14 for more details.

`clone_ingress_pkt_to_egress(clone_spec [ , field_list ] )`

Summary

Generate a copy of the original packet and submit it to the Buffering Mechanism.

Parameters

`clone_spec`	(VAL) An opaque identifier indicating additional run time characteristics of the clone operation.
`field_list`	(FLDLIST) An optional list of metadata field references.

Description

This action indicates that the switch should generate a copy of the original packet. The clone's Parsed Representation will match the original's immediately after parsing, with the exception that the fields listed in `field_list` are replaced with the original packet's values after being processed by the ingress pipeline.

The clone of the packet is submitted directly to the Buffering Mechanism as an independent packet instance. It does not go through ingress match+action processing.

The original packet continues to be processed as though the clone had not been produced.

The `clone_spec` is used to allow the configuration of other target specific characteristics of the clone operation as described in `clone_ingress_pkt_to_ingress`. In addition to other session attributes, `clone_spec` determines the egress specification (standard metadata `egress_spec`) that is presented to the Buffering Mechanism.

The cloned instance will have `instance_type` set to indicate that it is an egress clone.

The function may also be referred to as `clone_i2e`.

See the Section 14 for more details.

`clone_egress_pkt_to_egress(clone_spec [ , field_list ] )`

Summary

Duplicate the egress version of the packet and submit it to the Buffering Mechanism.

Parameters

`clone_spec`	(VAL) An opaque identifier indicating additional run time characteristics of the clone operation.
`field_list`	(FLDLIST) An optional list of metadata field references.

Description

The packet is marked for cloning at egress. Once the original packet completes the egress pipeline, the packet and its Parsed Representation of packet headers (including all modifications due to match+action) along with the metadata fields specified in `field_list`are submitted to the Buffering Mechanism as a new packet instance.

The original packet is forwarded as normal.

The `clone_spec`is used to allow the configuration of other target specific characteristics of the clone operation as described in `clone_ingress_pkt_to_ingress`. In addition to other session attributes, `clone_spec`determines the egress specification (standard metadata `egress_spec`) that is presented to the Buffering Mechanism.

The cloned instance will have `instance_type`set to indicate that it is an egress clone.

The function may also be referred to as `clone_e2e`.

See the Section 14 for more details.

9.1.1 Field Assignment and Saturation Attributes

Recall that a field may have sign and saturation attributes in its declaration. The logical behavior of the add_to_field is determined by these attributes as shown in the following logic for an unsigned field.

```
tmp = field + value
if (field.saturating && tmp < field.min)
    field = field.min
else if (field.saturating && tmp > field.max)
    field = field.max
else
    field = tmp % 2^{field.width}
```

where:

- `field.saturating`: boolean value indicating that the field is saturating.
- `field.min`: minimum allowed value determined by the field's bit width and signedness
- `field.max`: maximum allowed value determined by the field's bit width and signedness
- `field.width`: bit width of field

9.1.2 Parameter Binding

In several primitive actions above, a parameter may take one of:

- An immediate value; or
- A value from a table entry's action parameter data; or
- A reference to a field instance whose current value is used; or
- A reference to a register cell whose current value is used.

The P4 language does not specify limits on the specification of which of these may be exercised at a given time. However, it should be noted that there is a qualitative difference (in the sense that it imposes different functional requirements on the underlying target) between specifying a particular field instance in a P4 program and allowing a run time API to specify the field instance to reference when the table entry is added.

This is a binding-time issue; the first binds the field reference at compile time while the second allows run time binding. Targets may impose constraints on the flexibility allowed for such parameter binding. The difference must also be reflected in the run time interfaces that are generated.

9.2 Action Definitions

Actions are declared as functions.

```
action_function_declaration ::=
    action action_header { action_statement + }

action_header ::= action_name "(" [ param_list ] ")"
action_statement ::= action_name "(" [ arg [, arg]* ] ")" ;
arg ::= param_name | field_value | field_ref | header_ref
```

Parameter references to counters, meters or registers are by index and so pass through the `param_name` or `field_value` arguments.

Action function declarations must obey the following conventions:

- All parameters specified in the `action_header` are required. Optional parameters are *not* supported.
- The body of the function contains only:
- Calls to primitive actions.
- Calls to other action functions.
- Recursion is not allowed.

In the following example, the parameters `dst_mac`, `src_mac` and `vid` would be exposed via a run time API for adding entries to the table which used this action. The values passed to that API would then be set in the table entry being added so that they could be passed to this action for packets that hit that entry.

```
action route_ipv4(dst_mac, src_mac, vid) {
    modify_field(ethernet.dst_addr, dst_mac);
    modify_field(ethernet.src_addr, src_mac);
    modify_field(vlan_tag.vid, vid);
    add_to_field(ipv4.ttl, -1);
}
```

9.2.1 Parallel and Serial Semantics

In any instruction execution model, identifying whether a set of instructions is executed in parallel or in sequence must be identified in order to determine the behavior of the system. As an example, consider the statements:

```
    modify_field(hdr.fieldA, 1);
    modify_field(hdr.fieldB, hdr.fieldA);
```

Supposing that `hdr.fieldA` started with a value of `0`, the question is: what value will `hdr.fieldB` have after this instruction set is executed? Will it be `0` or `1`? With sequential semantics, the first statement is completed, leaving `1` in `fieldA`; then the second instruction is executed, propagating the `1` to `fieldB`. With parallel semantics, both actions are started at the same time, so the evaluation of `hdr.fieldA` in the second instruction resolves to `0` (since it has not yet changed), and so `hdr.fieldB` receives the value `0`.

P4 assumes **parallel** semantics for the application of all the primitive actions executing as a result of a match in a given table. The execution of actions across different tables assumes **sequential** semantics where the sequence is determined by the control flow, described in Section 12.

10 Action profile declarations

In some instances, action parameter values are not specific to a match entry but could be shared between different entries. Some tables might even want to share the same set of action parameter values. This can be expressed in P4 with action profiles. Action profiles are declarative structures specifying a list of potential actions, and possibly other attributes.

Entries are inserted at run time to specify the single action to be run if that entry is chosen - among the candidates included in the action profile declaration-, as well as the action parameter values to use.

Instead of statically binding one particular action profile entry to each match entry, one might want to associate multiple action profile entries with a match entry and let the system (i.e., data plane logic) dynamically bind one of the action profile entries to each class of packets. The `dynamic_action_selection` attribute enables such behavior. When `dynamic_action_selection` is specified, action profile entries can be bundled into groups by the run time, and a match entry can then tied to a group of action profile entries. To dictate a specific data-plane mechanism that chooses a particular action profile entry in a group, one should provide an action selector. An action selector chooses a particular action profile entry for each packet by either pseudo-randomly or predictably deriving a decision from header fields and/or metadata.

Here is the BNF for an action profile declaration:

```
action_profile_declaration ::=
    action_profile action_profile_name {
        action_specification
        [ size : const_value ; ]
        [ dynamic_action_selection : selector_name ; ]
    }

action_specification ::=
    actions { [ action_name ] + }

action_selector_declaration ::=
    action_selector selector_name {
        selection_key : field_list_calculation_name ;
    }
```

Action profiles are declared and applied with the following conventions:

- The size attribute indicates the number of entries required for the action profile. If this cannot be supported, an error will be signaled when the declaration is processed. If this attribute is omitted, there is no guarantee as to the number of entries that the action profile will be able to accomodate at run time.

11 Table Declarations

Tables are declarative structures specifying match and action operations, and possibly other attributes. The action specification (or action profile specification) in a table indicates which action functions are available to this table's entries.

Note that masks may be specified for fields in the table declaration. This should not be confused with masks for ternary matches. The masks specified in table declarations are statically applied to fields before the start of the match process. This allows arbitrary subfields to be used in exact match tables. This is intended for exact match tables; it is allowed for ternary matches in the syntax, though it is functionally redundant.

Match semantics are always the conjunction (AND) of all field match specifications.

Here is the BNF for a table declaration:

```
table_declaration ::=
    table table_name {
        [ reads { field_match + } ]
        table_actions
        [ min_size : const_value ; ]
        [ max_size : const_value ; ]
        [ size : const_value ; ]
        [ support_timeout : true | false ; ]
    }

field_match ::= field_or_masked_ref : field_match_type ;
field_or_masked_ref ::=
    header_ref | field_ref | field_ref mask const_value

field_match_type ::= exact | ternary | lpm | range | valid

table_actions ::=
```

```
    action_specification | action_profile_specification

action_specification ::=
    actions { [ action_name ] + }

action_profile_specification ::=
    action_profile : action_profile_name
```

This example is from the mTag edge switch program. It maps the packet's L2 destination to an mTag. If it fails to find a map, it may copy the packet to the CPU.

```
// Check if the packet needs an mtag and add one if it does.
table mTag_table {
    reads {
        ethernet.dst_addr    : exact;
        vlan.vid             : exact;
    }
    actions {
        add_mTag;     // Action called if pkt needs an mtag.
        common_copy_pkt_to_cpu; // If no mtag, send to the CPU
        no_op;
    }
    max_size                 : 20000;
}
```

For an implementation of ECMP using an action profile with an action selector, please see 15.8.3.

Match types have the following meanings.

- `exact`: The field value is matched against the table entry and the values must be identical for the entry to be considered.
- `ternary`: A mask provided with each entry in the table. This mask is ANDed with the field value before a comparison is made. The field value and the table entry value need only agree on the bits set in the entry's mask. Because of the possibilities of overlapping matches, a priority must be associated with each entry in a table using ternary matches.
- `lpm`: This is a special case of a ternary match. Each entry's mask selects a prefix by having a divide between 1s in the high order bits and 0s in the low order bits. The number of 1 bits gives the length of the prefix which is used as the priority of the entry.

- `range`: Each entry specifies a low and high value for the entry and the field matches only if it is in this range. Range end points are inclusive. Signedness of the field is used in evaluating the order.
- `valid`: Only applicable to packet header fields or header instances (not metadata fields), the table entry must specify a value of true (the field is valid) or false (the field is not valid) as match criteria.

Tables are defined and applied with the following conventions:

- Header references for matching may only be used with the `valid` match type.
- Exactly one of the actions indicated in either the `action_specification` or the `action_profile_specification` will be run when a table processes a packet.
  - Entries are inserted at run time and each rule specifies the single action to be run if that entry is matched.
  - Actions in the list may be primitive actions or compound actions.
- At run time, the table entry insert operation (not part of P4) must specify:
  - Values for each field specified in the `reads` entry.
  - The name of the action from the `action_specification` or the `action_profile_specification` and the parameters to be passed to the action function when it is called.
- A default action is taken when no table entry matches. This action is specified at run time. If no default action is specified and no entry matches, the table does not affect the packet and processing continues according to the imperative control flow.
- If `reads` is not present, the table will always execute the default action. If no default action has been specified, the table has no effect on the packet.
- The keyword `mask` may be used for a field to indicate that only the indicated bits should be used in the match. This mask is applied once to the Parsed Representation's field prior to any comparisons (compared to the per-entry mask which may differ from entry to entry).
- The match type `valid` indicates that the field's parent header (or, in the case of metadata, the field itself) should be tested for validity. The value of `1` will match when the header is valid; `0` will match when the header is not valid. As a reminder, the table does not have to explicitly include a match on a field's validity to safely match on its value - invalid fields will never match on a table entry that includes it. Note that metadata fields are *always* valid.
- The `min_size` attribute indicates the minimum number of entries required for the table. If this cannot be supported, an error will be signaled when the declaration

is processed.

- The `max_size` attribute is an indication that the table is not expected to grow larger than this size. If, at run time, the table has this many entries and another insert operation applied, it may be rejected.
- The `size` attribute is equivalent to specifying `min_size` and `max_size` with the same value.
- Although `size` and `min_size` are optional, failing to specify at least one of them may result in the table being eliminated as the compiler attempts to satisfy the other requirements of the program.
- The `support_timeout` attribute is used to enable ageing on a table. It is optional and its default value is `false`.

A no-op primitive action, `no_op`, is defined in P4 in Section 9.1. It may be used to indicate that a match should result in no change to the packet.

12 Packet Processing and Control Flow

A packet is processed by a sequence of match+action tables. At configuration time, the control flow (in what order the tables are to be applied) may be expressed with an imperative program. The imperative program may apply tables, call other control flow functions or test conditions.

The execution of a table is indicated with the `apply` instruction. The `apply` instruction itself can affect the control flow to which the packet is subject by specifying a set of control blocks from which one is selected to be executed. The choice of which block is selected may be determined by the action used on the packet or by whether a match was found at all.

The `apply` instruction has three modes of operation.

- Sequential: Control flow moves to the next statement unconditionally.
- Action Selection: The action that was applied to the packet determines the block of instructions to execute.
- Hit/Miss Check: Whether or not a match was found determines the block of instructions to execute.

Examples of each mode are given below, following the BNF. In conjunction with the if-else statement, this provides the mechanism for expressing control flow.

```
control_function_declaration ::=
    control control_fn_name control_block
```

```
control_block ::= { control_statement * }
control_statement ::=
    apply_table_call |
    apply_and_select_block |
    if_else_statement |
    control_fn_name ( ) ;

apply_table_call ::= apply ( table_name ) ;

apply_and_select_block ::= apply ( table_name ) { [ case_list ] }
case_list ::= action_case + | hit_miss_case +
action_case ::= action_or_default control_block
action_or_default ::= action_name | default
hit_miss_case ::= hit_or_miss control_block
hit_or_miss ::= hit | miss

if_else_statement ::=
    if ( bool_expr ) control_block
    [ else_block ]

else_block ::= else control_block | else if_else_statement
bool_expr ::=
    valid ( header_ref ) | bool_expr bool_op bool_expr |
    not bool_expr | ( bool_expr ) | exp rel_op exp | true | false

exp ::=
    exp bin_op exp | un_op exp | field_ref |
    value | ( exp )

bin_op ::= "+" | "*" | - | "<<" | ">>" | \& | "|" | ^
un_op ::= ~ | -
bool_op ::= or | and
rel_op ::= > | >= | == | <= | < | !=
```

In many cases, it is convenient to specify a sequence of fields. For example,

Operator precedence and associativity follows C programming conventions. As described in Section 4.2, the parser returns the name of the control function to begin match+action processing. When that function completes, the packet is passed to the queuing mechanism (unless the packet is discarded). The function `egress`, if defined, is called when a packet is dequeued. Then the packet is transmitted to a specific port indicated by the `egress_port` field in the metadata.

Tables are invoked on the packet with the `apply` operator as described at the beginning of this section. If the same table is invoked in multiple places from the control flow, those invocations all refer to the same table instance; that is, there is only one set of statistics, counters, meters, and match+action entries for the table. Targets may impose limitations on these table invocations such as disallowing recursion, only allowing tables to be referenced once, or only allowing control flow functions to be referenced once.

The simplest control flow is to execute a sequence of tables with the `apply` operator.

```
// The ingress control function
control ingress {
    // Verify mTag state and port are consistent
    apply(check_mtag);
    apply(identify_port);
    apply(select_output_port);
}
```

The `apply` operator can be used to control the instruction flow based on whether a match was found in the table. This is done by specifying a block enclosed in braces following the apply operation with `hit` and/or `miss` as the case selection labels. The mTag edge program includes the following example:

```
apply(egress_meter) {
    hit { // If egress meter table matched, apply policy
        apply(meter_policy);
    }
}
```

Alternatively, the `apply` operator can control the instruction flow based on the action applied by the table to the packet. Here is an example.

```
apply(routing_table) {
    ipv4_route_action { // IPv4 action was used
        apply(v4_rpf);
        apply(v4_acl);
    }
    ipv6_route_action { // IPv6 action was used
        apply(v6_option_check);
        apply(v6_acl);
    }
    default { // Some other action was used
        if (standard_metadata.ingress_port == 1) {
```

```
            apply(cpu_ingress_check);
        }
    }
}
```

Note that the two modes (match selection versus action selection) cannot be intermixed. They are differentiated due to the fact that `hit` and `miss` are reserved words and cannot be used as action function names.

13 Egress Port Selection, Replication and Queuing

In P4, the `egress_spec` metadata field is used to specify the destination or destinations of a packet. In addition, for devices supporting priority queuing, `egress_spec` may indicate the queue associated with each destination. An `egress_spec` value may represent a physical port, a logical port (e.g., a tunnel, a LAG, a route, or a VLAN flood group), or a multicast group.

P4 assumes that the Buffering Mechanism implements a function that maps `egress_spec` to a collection of packet instances represented as triples:

(`packet`, `egress_port`, `egress_instance`).

The Buffering Mechanism is responsible for generating each packet instances along with these metadata fields and sending it as necessary to reach its egress port through the egress match+action tables.

This mapping of `egress_spec` values to sets of packet instances is currently outside the scope of P4; a forwarding element may statically map values to destinations or may allow configuration of the map through a management interface. The run time table programing interfaces must have access to this information to properly program the tables declared in the P4 program.

The flow of packets through a forwarding element is as follows. Recall that, as depicted in Figure 1, processing is divided between ingress and egress with the packet possibly being buffered between the two. The parser normally terminates by indicating the control function used to begin processing. Upon completion of that control function, the packet is submitted to the buffering system.

The buffers are assumed to be organized into one or more queues per egress port. The details of queue structure and dequeuing disciplines is considered to be target specific, though targets may use P4 to expose configuration (and even to define actions resulting from data plane events) related to queuing behavior.

A single copy of each packet traverses the Ingress Pipeline. At the completion of ingress processing, the switch determines the queue(s) to place the packet in based upon the `egress_spec` value. A packet that is sent to multiple destinations may be placed in multiple queues.

When the packet is dequeued, it is processed in the Egress Pipeline by the control function `egress`. A separate copy of the packet is sent through the Egress Pipeline for each destination, requiring the Buffering Mechanism to replicate the packet. The physical egress port is known at the time the packet is dequeued; this value is passed through the Egress Pipeline as an immutable metadata field named `egress_port`. To support multiple copies of packets being sent to the same physical port (e.g., sending to multiple VLANs on one port), the immutable metadata field `egress_instance` contains a unique value for each copy. The semantics of `egress_instance` are target specific.

14 Recirculation and Cloning

Many standard networking functions, such as mirroring and recursive packet processing, require more complicated primitives than setting or testing fields. To support such operations, P4 provides primitive actions that allow a packet to be recirculated (sent back to the start of the processing pipeline) or cloned (a second instance of the packet is created).

Note that cloning is not intended to be the mechanism by which multicast is normally implemented. That is expected to be done by the Buffering Mechanism in conjunction with the egress specification. See Section 13.

Here is a table that summarizes the different operations. The first four (clone) operations create an entirely new instance of the packet. The last two, `resubmit` and `recirculate`, operate on the original packet and do not, by themselves, result in the generation of a new packet.

Name	**Source**	**Insertion Point**
`clone_ingress_pkt_to_ingress`	Original ingress pkt	Ingress parser
`clone_egress_pkt_to_ingress`	Post deparsed pkt	Ingress parser
`clone_ingress_pkt_to_egress`	Original ingress pkt	Buffering Mechanism
`clone_egress_pkt_to_egress`	Post deparsed pkt	Buffering Mechanism
`resubmit`	Original ingress pkt	Ingress parser
`recirculate`	Post deparsed pkt	Ingress parser

Table 6: Clone and Recirculation Primitives

14.1 Clone

The clone operations generate a new version of the packet. The original version continues to be processed as if the clone operation did not take place. We use the term clone (rather than mirror) to emphasize that this action is only responsible for generating a new version of the packet. Mirroring requires additional configuration. The clone mechanism may have additional applications.

The source of the clone may be the original instance of the packet (an ingress clone), or the packet as it would exit the switch (an egress clone). The processing of the new instance may be limited to the egress pipeline ("to egress") or it may start with the ingress pipeline ("to ingress"). Hence we have four different clone operations.

For cloned packets, the `instance_type` metadata field is used to distinguish between the original and cloned packet instances.

If multiple clone actions are executed on one packet, that many clone instances should be generated. However, specific targets may impose limits on the number of clone instances supported.

14.1.1 Clone to Ingress

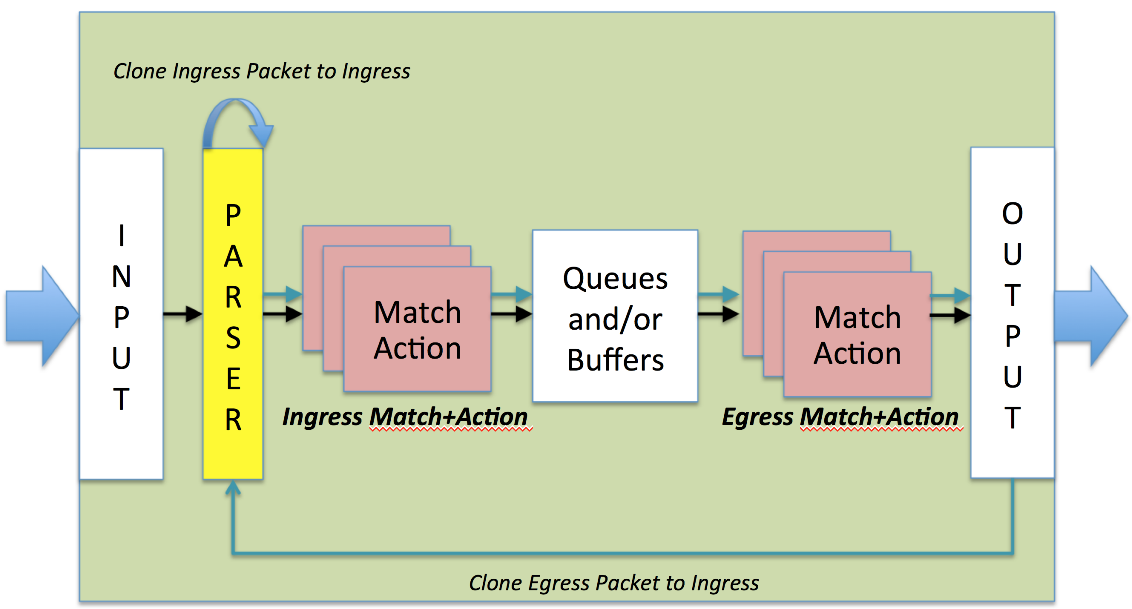

Figure 3: Cloning to Ingress, from Ingress or Egress

Figure 3 shows the paths for a cloned packet submitted to the ingress. The source may be from the ingress itself, indicating that a copy of the original packet is given to the parser, or from the egress, in which case a copy of the packet as it is transmitted is created and submitted to the parser.

14.1.2 Clone to Egress

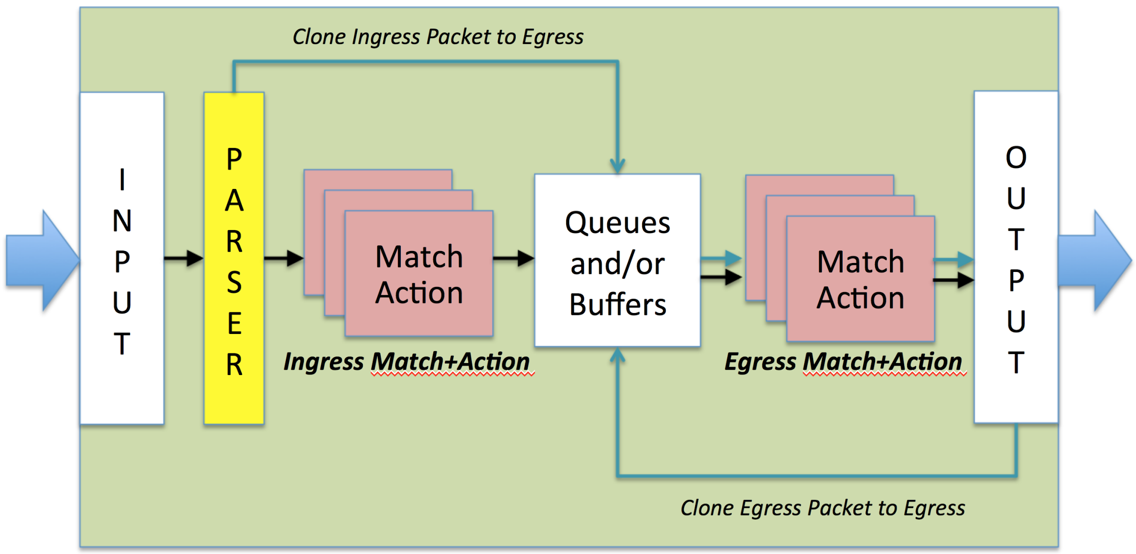

Figure 4: Cloning to Egress, from Ingress or Egress

Figure 4 shows the paths for a cloned packet submitted to the egress pipeline. The source may be from the ingress, indicating that a copy of the original packet as parsed is submitted to the Buffering Mechanism; or the source may be from the egress, in which case a copy of the packet (and some of its Parsed Representation) just prior to deparsing is created and submitted to the Buffering Mechanism.

Since the Buffering Mechanism requires an egress specification (`metadata.egress_spec`) to determine how to handle the packet, an egress specification should be associated with the `clone_spec` associated with the instance by the primitive operation. In fact, the `clone_spec` could simply be an `egress_spec` for some targets.

14.1.3 Mirroring

Mirroring, or port monitoring, is a standard networking function described, for example, at http://en.wikipedia.org/wiki/Port_mirroring. In this section we describe one approach to implementing mirroring with P4.

Mirroring involves the following:

- Identifying the packets to be mirrored.
- Generating the mirrored instances of those packets
- Specifying what actions should be done on the mirrored instances

Normally, these functions are logically grouped together into a *mirror session*.

Assuming minimal additional target support (for example, a target might provide intrinsic metadata that would directly execute everything necessary for mirroring) a P4 program might include the following to support ingress mirroring of packets which are selected based on a combination of ingress port, VLAN ID, L3 addresses and IP protocol.

In this example, the Buffering Mechanism is assumed to provide a programmable map from the `clone_spec` parameter passed to `clone_i2e` to an `egress_port` number.

First, a table that matches on these characteristics would be declared. It would reference an action like the following:

```
action mirror_select(session) { // Select packets; map to session
    modify_field(local_metadata.mirror_session, session);
    clone_i2e(session, mirror_fld_list);
}
```

where

```
field_list mirror_field_list {
    local_metadata.mirror_session;
}
```

indicates that the mirror session must be preserved in the cloned packet.

This action results in a new copy of the ingress packet to be submitted to the egress. The run time APIs allow the specification of exactly which packets get mirrored. They also have the flexibility to select the mirror session ID associated with each such packet. The `mirror_select` table would be introduced into the control flow for the ingress pipeline, probably early in processing.

A table matching on `local_metadata.mirror_session` would be introduced in the egress pipeline. Assume a value of 0 means "not mirrored", so the table could be applied to all packets but only select the actions related to mirroring for those marked with a mirror session. This table would exercise an action like:

```
action mirror_execute(trunc_length) {
    truncate(trunc_length);
}
```

For this example, the only action taken is the truncation of the mirrored packet. However the function could include the data used for an encapsulation header allowing each mirror session to be sent to a different remote monitoring session. The encapsulation header values would be programmed at run time.

Egress mirroring would follow a similar pattern with the primary difference being the primitive action used would be `clone_e2e`.

14.2 Resubmit and Recirculate

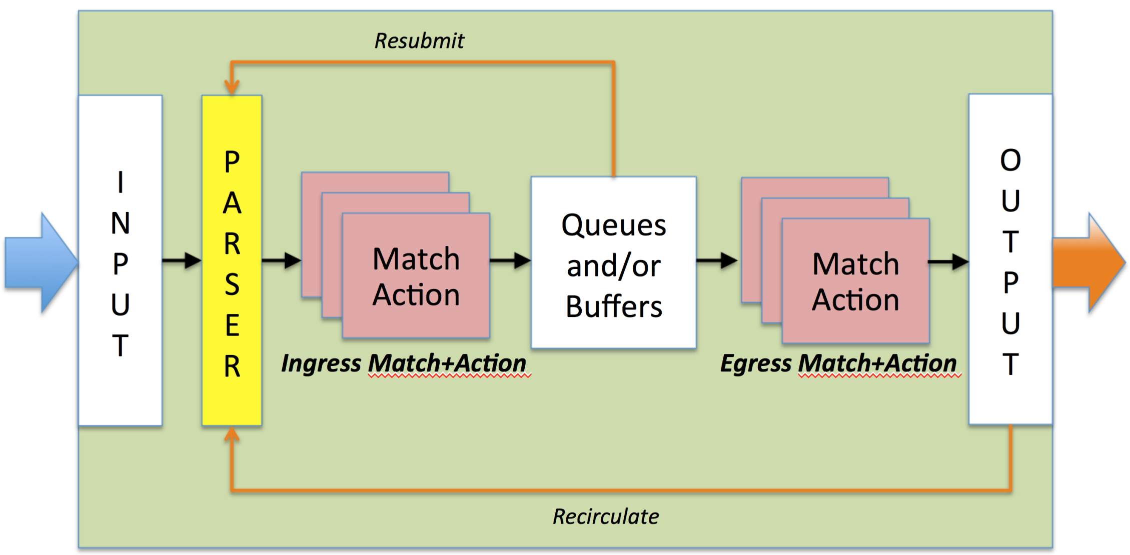

Figure 5: Resubmit and Recirculate

Figure 5 shows the paths for resubmitting a packet to the parser for processing. The top path shows a resubmit process. The `resubmit` action is signalled in the ingress pipeline. Upon completing that pipeline, the **original** packet seen on ingress is resubmitted to the parser along with additional metadata as specified by the action. The parser may use the new metadata to make different parsing decisions than on the original pass through the parser.

The lower path shows the path for recirculation. After the packet has completed both ingress and egress processing, it is deparsed and sent back to the parser. The new packet is reparsed, possibly with metadata preserved from the original packet, and passed to the ingress pipeline as usual.

For `resubmit` and `recirculate`, the `instance_type` metadata field distinguishes between first and later times the packet is being processed.

15 Appendices

15.1 Errata

- The syntax of action lists in table declarations is inconsistent. The specification implies these are space-separated lists. Semicolon-separated lists would be consistent.
- The syntax of counter references is inconsistent compared to meters. Meters use bracket notation while counters use a separate parameter.
- The mechanism to refer to the output of a meter is over-specified. The output of a meter (the metadata field into which the "color" returned by a meter is stored) is allowed to be specified both in the declaration of the meter as well as when the meter is invoked.

15.2 Programming Conventions (Incomplete)

The following is a list of conventions suggested for P4 programs.

- Parsing begins with the parser state function named `start`.
- Control flow begins with the control function `ingress`.

15.3 Revision History

Release	Release Date	Summary of Changes
1.0.0-rc1	2014-09-08	First public version.
1.0.0-rc2	2014-09-09	Minor typos.
1.0.0-rc3	2014-12-30	Fixed some missing tildes (negations). Drop in parser is now `parser_drop`. Added `add` primitive action. Added errata section.
1.0.1	2015-01-28	Added action profiles and action selectors. Added attribute `support_timeout` to tables.
1.0.2	2015-03-03	Added `push` and `pop` primitive actions.

Table 7: Revision History

15.4 Terminology (Incomplete)

Term	Definition
Control Flow	The logic that selects which tables are applied to a packet when it is processed by a pipeline. Used to resolve order dependencies.
Egress Queuing	An abstract P4 functional block logically separating ingress and egress processing. Implementations may expose queuing and buffer resource management interfaces for this block, but this not specified by P4.
Egress Specification	Metadata set by the ingress pipeline which determines the set of destination ports (and number of instances on each port) to which the packet should be sent
Order Dependency	A sequence of match and action operations whose result depends on the order of execution. For example, one table may set a field which another table uses for a match. The control flow is used to determine which of the possible effects is intended.
Parsed Representation	A representation of a packet's header as a set of header instances, each of which is composed of fields.
Parser	A functional block which maps a packet to a Parsed Representation
Pipeline	A sequence of match+action tables. run time When a switch is processing packets. This is distinguished from configure time, though these operations may occur at the same time in some implementations.

Table 8: Terminology

15.5 Summary of P4 BNF

```
p4_program ::= p4_declaration +

p4_declaration ::=
    header_type_declaration |
    instance_declaration |
    field_list_declaration |
    field_list_calculation_declaration |
    calculated_field_declaration |
    value_set_declaration |
    parser_function_declaration |
    parser_exception_declaration |
```

```
    counter_declaration |
    meter_declaration |
    register_declaration |
    primitive_action_declaration |
    action_function_declaration |
    action_profile_declaration |
    action_selector_declaration |
    table_declaration |
    control_function_declaration

const_value ::= [ "+" | - ] [ width_spec ] unsigned_value
unsigned_value ::= binary_value | decimal_value | hexadecimal_value

binary_value ::=  binary_base binary_digit+
decimal_value ::= decimal_digit+
hexadecimal_value ::= hexadecimal_base hexadecimal_digit+

binary_base ::= 0b | 0B
hexadecimal_base ::= 0x | 0X

binary_digit ::= _ | 0 | 1
decimal_digit ::= binary_digit | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hexadecimal_digit ::=
    decimal_digit | a | A | b | B | c | C | d | D | e | E | f | F

width_spec ::= decimal_digit+ '
field_value ::= const_value
header_type_declaration ::=
   header_type header_type_name { header_dec_body }

header_dec_body ::=
    fields { field_dec + }
    [ length : length_exp ; ]
    [ max_length : const_value ; ]

field_dec ::= field_name : bit_width [ ( field_mod ) ];
field_mod ::= signed | saturating | field_mod , field_mod
length_bin_op ::= "+" | - | "*" | "<<" | ">>"
length_exp ::=
    const_value |
    field_name |
    length_exp length_bin_op length_exp |
```

```
    ( length_exp )

bit_width ::= const_value | "*"
instance_declaration ::= header_instance | metadata_instance
header_instance ::= scalar_instance | array_instance
scalar_instance ::= header header_type_name instance_name ;
array_instance ::=
    header header_type_name
        instance_name "[" const_value "]" ;

metadata_instance ::=
   metadata header_type_name
       instance_name [ metadata_initializer ] | ;

metadata_initializer ::= { [ field_name : field_value ; ] + }
header_ref ::= instance_name | instance_name "[" index "]"
index ::= const_value | last
field_ref ::= header_ref . field_name
field_list_declaration ::=
    field_list field_list_name {
        [ field_list_entry ; ] +
    }

field_list_entry ::=
    field_ref | header_ref | field_value | field_list_name | payload
field_list_calculation_declaration ::=
    field_list_calculation field_list_calculation_name {
        input {
            [ field_list_name ; ] +
        }
        algorithm : stream_function_algorithm_name ;
        output_width : const_value ;
    }
calculated_field_declaration ::=
    calculated_field field_ref { update_verify_spec + }

update_verify_spec ::=
    update_or_verify field_list_calculation_name [ if_cond ] ;

update_or_verify ::= update | verify
if_cond ::= if ( calc_bool_cond )
calc_bool_cond ::=
```

```
    valid ( header_ref | field_ref ) |
    field_ref == field_value
value_set_declaration ::= parser_value_set value_set_name;
parser_function_declaration ::=
    parser parser_state_name { parser_function_body }

parser_function_body ::=
    extract_or_set_statement*
    return_statement

extract_or_set_statement ::= extract_statement | set_statement
extract_statement ::= extract ( header_extract_ref );
header_extract_ref ::=
     instance_name |
     instance_name "[" header_extract_index "]"

header_extract_index ::= const_value | next

set_statement ::= set_metadata ( field_ref, metadata_expr ) ;
metadata_expr ::= field_value | field_or_data_ref

return_statement ::=
    return_value_type |
    return select ( select_exp ) { case_entry + }

return_value_type ::=
    return parser_state_name ; |
    return control_function_name ; |
    parse_error parser_exception_name ;

case_entry ::= value_list : case_return_value_type ;
value_list ::= value_or_masked [ , value_or_masked ]* | default

case_return_value_type ::=
    parser_state_name |
    control_function_name |
    parse_error parser_exception_name

value_or_masked ::=
    field_value | field_value mask field_value | value_set_name

select_exp ::= field_or_data_ref [, field_or_data_ref] *
```

```
field_or_data_ref ::=
    field_ref |
    latest.field_name |
    current( const_value , const_value )
parser_exception_declaration ::=
    parser_exception parser_exception_name {
        set_statement *
        return_or_drop ;
    }

return_or_drop ::= return_to_control | parser_drop
return_to_control ::= return control_function_name
counter_declaration ::=
    counter counter_name {
        type : counter_type ;
        [ direct_or_static ; ]
        [ instance_count : const_value ; ]
        [ min_width : const_value ; ]
        [ saturating ; ]
    }

counter_type ::= bytes | packets
direct_or_static ::= direct_attibute | static_attribute
direct_attribute ::= direct : table_name
static_attribute ::= static : table_name
meter_declaration ::=
    meter meter_name {
        type : meter_type ;
        result : field_ref ;
        [ direct_or_static ; ]
        [ instance_count : const_value ; ]
    }

meter_type ::= bytes | packets
register_declaration ::=
    register register_name {
        width_or_layout ;
        [ direct_or_static ; ]
        [ instance_count : const_value ; ]
        [ attribute_list ; ]
    }
```

```
width_or_layout ::= width_declaration | layout_declaration
width_declaration ::= width : const_value ;
layout_declaration ::= layout : header_type_name ;

attribute_list ::= attributes : attr_entry
attr_entry ::= signed | saturating | attr_entry , attr_entry
register_reference ::=
    register_name "[" const_value "]" [.field_name]
primitive_action_declaration ::=
    primitive_action action_name "(" [ param_list ] ")" ;

param_list ::= param_name [, param_name]*
action_function_declaration ::=
    action action_header { action_statement + }

action_header ::= action_name "(" [ param_list ] ")"
action_statement ::= action_name "(" [ arg [, arg]* ] ")" ;
arg ::= param_name | field_value | field_ref | header_ref

action_profile_declaration ::=
    action_profile action_profile_name {
        action_specification
        [ size : const_value ; ]
        [ dynamic_action_selection : selector_name ; ]
    }

action_specification ::=
    actions { [ action_name ] + }

action_selector_declaration ::=
    action_selector selector_name {
        selection_key : field_list_calculation_name ;
    }


table_declaration ::=
    table table_name {
        [ reads { field_match + } ]
        table_actions
        [ min_size : const_value ; ]
        [ max_size : const_value ; ]
        [ size : const_value ; ]
```

```
        [ support_timeout : true | false ; ]
    }

field_match ::= field_or_masked_ref : field_match_type ;
field_or_masked_ref ::=
    header_ref | field_ref | field_ref mask const_value

field_match_type ::= exact | ternary | lpm | range | valid

table_actions ::=
    action_specification | action_profile_specification

action_specification ::=
    actions { [ action_name ] + }

action_profile_specification ::=
    action_profile : action_profile_name

control_function_declaration ::=
    control control_fn_name control_block
control_block ::= { control_statement * }
control_statement ::=
    apply_table_call |
    apply_and_select_block |
    if_else_statement |
    control_fn_name ( ) ;

apply_table_call ::= apply ( table_name ) ;

apply_and_select_block ::= apply ( table_name ) { [ case_list ] }
case_list ::= action_case + | hit_miss_case +
action_case ::= action_or_default control_block
action_or_default ::= action_name | default
hit_miss_case ::= hit_or_miss control_block
hit_or_miss ::= hit | miss

if_else_statement ::=
    if ( bool_expr ) control_block
    [ else_block ]

else_block ::= else control_block | else if_else_statement
bool_expr ::=
```

```
    valid ( header_ref ) | bool_expr bool_op bool_expr |
    not bool_expr | ( bool_expr ) | exp rel_op exp | true | false

exp ::=
    exp bin_op exp | un_op exp | field_ref |
    value | ( exp )

bin_op ::= "+" | "*" | - | "<<" | ">>" | \& | "|" | ^
un_op ::= ~ | -
bool_op ::= or | and
rel_op ::= > | >= | == | <= | < | !=
```

15.6 P4 Reserved Words

The following are reserved words in P4 and should not be used as identifiers.[1]

```
apply
current
default
else
hit
if
last
latest
parse_error
payload
select
switch
```

15.7 Field Value Conversions

As mentioned in Section 1.5.1, values may need to be converted when used in an expression or assigned to a field instance. The conversion will depend the the source and destination widths and signedness, and whether the destination is saturating.

A value is signed if (1) it has an explicit minus ("-") preceding its representation; or (2) it is a field instance with the signed attribute in its declaration. Otherwise it is unsigned.

The rules for conversion are as follows:

[1]There is an open issue whether all P4 keywords will in fact be reserved.

- If the source and destinations have the same width, the binary value of the source is used, but the interpretation may change if the signedness is different.
  - Example: source is unsigned, 7 bits with a value of 127 and the dest is signed, 7 bits, the result will be interpreted as -1.
- If the source width is less than the destination width, the source is extended based on its own signedness.
  - Example: Source is signed, 7'b1111111 and dest is 8 bits; the result is 8'b11111111.
  - Example: Source is unsigned 4'b1100 and dest is 8 bits; the result is 8'b00001100.
- If the source width is greater than the destination width, the result depends on whether the destination is saturating. The effect should be the same as adding the value represented by the source to the destination when the destination is 0.
  - Example: Source is signed, and negative, destination is saturating. the result is 0.
  - Example: Source is unsigned, has value 17 and the destination is 4 bits, unsigned and saturating; the result is 15 as that is the saturated value of the destination.
  - Example: As above, but the destination is not saturating; the result is 1 as the destination would wrap above 15. This is equivalent to truncating the source.

For expressions, the value with largest bit width is identified and all other values are converted to this width according to their own signedness. The expression is then evaluated and the result is converted as necessary according to its use.

15.8 Examples

15.8.1 The Annotated mTag Example

This section presents the mTag example. The example describes two separate P4 programs, mtag-edge and mtag-aggregation, as described in the introduction in Section 1.2.

The code is written in P4 whose syntax allows the application of a C preprocessor to P4 files. Thus directives such as `#define` and `#include` are used in the program with the same effects as if writing C code. This is a convention used by these examples; the P4 language does not mandate this syntax.

The example code is split into the following files

- `headers.p4`: The declaration of all header types used in both programs.

- `parser.p4`: The parser program shared by both programs.
- `actions.p4`: Common actions used by both programs.
- `mtag-edge.p4`: The main program for the edge switch
- `mtag-aggregation.p4`: The main program for any aggregation switch

The full source for all files is provided on the P4 website [2].

We start with `header.p4`.

```
////////////////////////////////////////////////////////////////
// Header type definitions
////////////////////////////////////////////////////////////////

// Standard L2 Ethernet header
header_type ethernet_t {
    fields {
        dst_addr        : 48; // width in bits
        src_addr        : 48;
        ethertype       : 16;
    }
}

// Standard VLAN tag
header_type vlan_t {
    fields {
        pcp             : 3;
        cfi             : 1;
        vid             : 12;
        ethertype       : 16;
    }
}

// The special m-tag used to control forwarding through the
// aggregation layer of the data center
header_type mTag_t {
    fields {
        up1             : 8; // From edge to agg
        up2             : 8; // Up from lower agg to upper agg
        down1           : 8; // Down from upper agg to lower agg
        down2           : 8; // Back to edge from agg
        ethertype       : 16; // Ethertype of encapped packet
    }
```

```
}

// Standard IPv4 header
header_type ipv4_t {
    fields {
        version         : 4;
        ihl             : 4;
        diffserv        : 8;
        totalLen        : 16;
        identification  : 16;
        flags           : 3;
        fragOffset      : 13;
        ttl             : 8;
        protocol        : 8;
        hdrChecksum     : 16;
        srcAddr         : 32;
        dstAddr         : 32;
        options         : *;  // Variable length options
    }
    length : ihl * 4;
    max_length : 60;
}

// Assume standard metadata from compiler.

// Define local metadata here.
//
// copy_to_cpu is an example of target specific intrinsic metadata
// It has special significance to the target resulting in a
// copy of the packet being forwarded to the management CPU.

header_type local_metadata_t {
    fields {
        cpu_code        : 16; // Code for packet going to CPU
        port_type       : 4;  // Type of port: up, down, local...
        ingress_error   : 1;  // An error in ingress port check
        was_mtagged     : 1;  // Track if pkt was mtagged on ingr
        copy_to_cpu     : 1;  // Special code resulting in copy to CPU
        bad_packet      : 1;  // Other error indication
        color           : 8;  // For metering
    }
}
```

The parser function shared by the programs is as follows.

```
//////////////////////////////////////////////////////////////////
// Parser functions and related definitions
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// Header instance definitions
//
// Header instances are usually defined with the parser as
// that is where they are initialized.
//
//////////////////////////////////////////////////////////////////

header ethernet_t ethernet;
header vlan_t vlan;
header mTag_t mtag;
header ipv4_t ipv4;

// Local metadata instance declaration
metadata local_metadata_t local_metadata;

//////////////////////////////////////////////////////////////////
// Parser state machine description
//////////////////////////////////////////////////////////////////

// Start with ethernet always.
parser start {
    return ethernet;
}

parser ethernet {
    extract(ethernet);   // Start with the ethernet header
    return select(latest.ethertype) {
        0x8100:     vlan;
        0x800:      ipv4;
        default:    ingress;
    }
}

// Extract the VLAN tag and check for an mTag
parser vlan {
```

```
    extract(vlan);
    return select(latest.ethertype) {
        0xaaaa:     mtag;
        0x800:      ipv4;
        default:    ingress;
    }
}

// mTag is allowed after a VLAN tag only (see above)
parser mtag {
    extract(mtag);
    return select(latest.ethertype) {
        0x800:      ipv4;
        default:    ingress;
    }
}

parser ipv4 {
    extract(ipv4);
    return ingress;  // All done with parsing; start matching
}
```

Here are the common actions for the two programs.

```
////////////////////////////////////////////////////////////////
//
// actions.p4
//
// This file defines the common actions that can be exercised by
// either an edge or an aggregation switch.
//
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
// Actions used by tables
////////////////////////////////////////////////////////////////

// Copy the packet to the CPU;
action common_copy_pkt_to_cpu(cpu_code, bad_packet) {
    modify_field(local_metadata.copy_to_cpu, 1);
    modify_field(local_metadata.cpu_code, cpu_code);
    modify_field(local_metadata.bad_packet, bad_packet);
```

```
}

// Drop the packet; optionally send to CPU and mark bad
action common_drop_pkt(do_copy, cpu_code, bad_packet) {
    modify_field(local_metadata.copy_to_cpu, do_copy);
    modify_field(local_metadata.cpu_code, cpu_code);
    modify_field(local_metadata.bad_packet, bad_packet);
    drop();
}

// Set the port type; see run time mtag_port_type.
// Allow error indication.
action common_set_port_type(port_type, ingress_error) {
    modify_field(local_metadata.port_type, port_type);
    modify_field(local_metadata.ingress_error, ingress_error);
}
```

Here are excerpts from the edge program.

```
////////////////////////////////////////////////////////////////
//
// mtag-edge.p4
//
// This file defines the behavior of the edge switch in an mTag
// example.
//
//
////////////////////////////////////////////////////////////////

// Include the header definitions and parser
// (with header instances)
#include "headers.p4"
#include "parser.p4"
#include "actions.p4"  // For actions marked "common_"

#define PORT_COUNT 64  // Total ports in the switch

////////////////////////////////////////////////////////////////
// Table definitions
////////////////////////////////////////////////////////////////

// Remove the mtag for local processing/switching
```

```
action _strip_mtag() {
    // Strip the tag from the packet...
    remove_header(mtag);
    // but keep state that it was mtagged.
    modify_field(local_metadata.was_mtagged, 1);
}

// Always strip the mtag if present on the edge switch
table strip_mtag {
    reads {
        mtag     : valid; // Was mtag parsed?
    }
    actions {
        _strip_mtag;      // Strip mtag and record metadata
        no_op;            // Pass thru otherwise
    }
}

////////////////////////////////////////////////////////////////

// Identify ingress port: local, up1, up2, down1, down2
table identify_port {
    reads {
        standard_metadata.ingress_port : exact;
    }
    actions { // Each table entry specifies *one* action
        common_set_port_type;
        common_drop_pkt;        // If unknown port
        no_op;         // Allow packet to continue
    }
    max_size : 64; // One rule per port
}

. . .  // Removed code related to local switching

// Add an mTag to the packet; select egress spec based on up1
action add_mTag(up1, up2, down1, down2) {
    add_header(mtag);
    // Copy VLAN ethertype to mTag
    modify_field(mtag.ethertype, vlan.ethertype);

    // Set VLAN's ethertype to signal mTag
```

```
    modify_field(vlan.ethertype, 0xaaaa);

    // Add the tag source routing information
    modify_field(mtag.up1, up1);
    modify_field(mtag.up2, up2);
    modify_field(mtag.down1, down1);
    modify_field(mtag.down2, down2);

    // Set the destination egress port as well from the tag info
    modify_field(standard_metadata.egress_spec, up1);
}

// Count packets and bytes by mtag instance added
counter pkts_by_dest {
    type : packets;
    direct : mTag_table;
}

counter bytes_by_dest {
    type : bytes;
    direct : mTag_table;
}

// Check if the packet needs an mtag and add one if it does.
table mTag_table {
    reads {
        ethernet.dst_addr    : exact;
        vlan.vid             : exact;
    }
    actions {
        add_mTag;  // Action called if pkt needs an mtag.
        // Option: If no mtag setup, forward to the CPU
        common_copy_pkt_to_cpu;
        no_op;
    }
    max_size                 : 20000;
}

// Packets from agg layer must stay local; enforce that here
table egress_check {
    reads {
        standard_metadata.ingress_port : exact;
```

```
        local_metadata.was_mtagged : exact;
    }

    actions {
        common_drop_pkt;
        no_op;
    }
    max_size : PORT_COUNT; // At most one rule per port
}

// Egress metering; this could be direct, but we let SW
// use whatever mapping it might like to associate the
// meter cell with the source/dest pair
meter per_dest_by_source {
    type : bytes;
    result : local_metadata.color;
    instance_count : PORT_COUNT * PORT_COUNT;  // Per source/dest pair
}

action meter_pkt(meter_idx) {
    meter(per_dest_by_source, meter_idx, local_metadata.color);
}

// Mark packet color, for uplink ports only
table egress_meter {
    reads {
        standard_metadata.ingress_port : exact;
        mtag.up1 : exact;
    }
    actions {
        meter_pkt;
        no_op;
    }
    size : PORT_COUNT * PORT_COUNT;  // Could be smaller
}

// Apply meter policy
counter per_color_drops {
    type : packets;
    direct : meter_policy;
}
```

```
table meter_policy {
    reads {
        metadata.ingress_port : exact;
        local_metadata.color : exact;
    }
    actions {
        drop; // Automatically counted by direct counter above
        no_op;
    }
    size : 4 * PORT_COUNT;
}


////////////////////////////////////////////////////////////////
// Control function definitions
////////////////////////////////////////////////////////////////

// The ingress control function
control ingress {

    // Always strip mtag if present, save state
    apply(strip_mtag);

    // Identify the source port type
    apply(identify_port);

    // If no error from source_check, continue
    if (local_metadata.ingress_error == 0) {
        // Attempt to switch to end hosts
        apply(local_switching); // not shown; matches on dest addr

        // If not locally switched, try to setup mtag
        if (standard_metadata.egress_spec == 0) {
            apply(mTag_table);
        }
     }
}

// The egress control function
control egress {
    // Check for unknown egress state or bad retagging with mTag.
    apply(egress_check);
```

```
    // Apply egress_meter table; if hit, apply meter policy
    apply(egress_meter) {
        hit {
            apply(meter_policy);
        }
    }
}
```

The key table for mtag-aggregation is shown below.

```
////////////////////////////////////////////////////////////////
//
// mtag-aggregation.p4
//
////////////////////////////////////////////////////////////////

// Include the header definitions and parser (with header instances)
#include "headers.p4"
#include "parser.p4"
#include "actions.p4"  // For actions marked "common_"

////////////////////////////////////////////////////////////////
// check_mtag table:
//   Make sure pkt has mtag; Apply drop or to-cpu policy if not
////////////////////////////////////////////////////////////////

table check_mtag { // Statically programmed w/ one entry
. . . // Reads if mtag valid; drop or copy to CPU
}

////////////////////////////////////////////////////////////////
// identify_port table:
//   Check if up or down facing port as programmed at run time.
////////////////////////////////////////////////////////////////

table identify_port {
. . . // Read ingress_port; call common_set_port_type.
}

////////////////////////////////////////////////////////////////
```

```
// Actions to copy the proper field from mtag into the egress spec
action use_mtag_up1() { // This is actually never used on agg switches
    modify_field(standard_metadata.egress_spec, mtag.up1);
}
action use_mtag_up2() {
    modify_field(standard_metadata.egress_spec, mtag.up2);
}
action use_mtag_down1() {
    modify_field(standard_metadata.egress_spec, mtag.down1);
}
action use_mtag_down2() {
    modify_field(standard_metadata.egress_spec, mtag.down2);
}

// Table to select output spec from mtag
table select_output_port {
    reads {
        local_metadata.port_type  : exact; // Up, down, level 1 or 2.
    }
    actions {
        use_mtag_up1;
        use_mtag_up2;
        use_mtag_down1;
        use_mtag_down2;
        // If port type is not recognized, previous policy applied
        no_op;
    }
    max_size : 4; // Only need one entry per port type
}

////////////////////////////////////////////////////////////////
// Control function definitions
////////////////////////////////////////////////////////////////

// The ingress control function
control ingress {
    // Verify mTag state and port are consistent
    apply(check_mtag);
    apply(identify_port);
    apply(select_output_port);
}
```

```
// No egress function used in the mtag-agg example.
```

The following is an example header file that might be used with the mtag example above. This shows the following:

- Type definitions for port types (`mtag_port_type_t`) meter levels (`mtag_meter_levels_t`) and a table entry handle (`entry_handle_t`).
- An example function to add an entry to the `identify_port` table, `table_identify_port_add_with_set_port_type`. The action to use with the entry is indicated at the end of the function name: `set_port_type`.
- Functions to set the default action for the identify_port table: `table_indentify_port_default_common_drop_pkt` and `table_indentify_port_default_common_set_port_type`.
- A function to add an entry to the mTag table: `table_mTag_table_add_with_add_mTag`
- A function to get a counter associated with the meter table: `counter_per_color_drops_get`.

```
/**
 * Run time header file example for CCR mTag example
 */


#ifndef MTAG_RUN_TIME_H
#define MTAG_RUN_TIME_H

/**
 * @brief Port types required for the mtag example
 *
 * Indicates the port types for both edge and aggregation
 * switches.
 */

typedef enum mtag_port_type_e {
    MTAG_PORT_UNKNOWN,         /* Uninitialized port type */
    MTAG_PORT_LOCAL,           /* Locally switch port for edge */
    MTAG_PORT_EDGE_TO_AG1,     /* Up1: edge to agg layer 1 */
    MTAG_PORT_AG1_TO_AG2,      /* Up2: Agg layer 1 to agg layer 2 */
    MTAG_PORT_AG2_TO_AG1,      /* Down2: Agg layer 2 to agg layer 1 */
    MTAG_PORT_AG1_TO_EDGE,      /* Down1: Agg layer 1 to edge */
    MTAG_PORT_ILLEGAL,         /* Illegal value */
```

```
    MTAG_PORT_COUNT
} mtag_port_type_t;

/**
 * @brief Colors for metering
 *
 * The edge switch supports metering from local ports up to the
 * aggregation layer.
 */

typedef enum mtag_meter_levels_e {
    MTAG_METER_COLOR_GREEN,  /* No congestion indicated */
    MTAG_METER_COLOR_YELLOW, /* Above low water mark */
    MTAG_METER_COLOR_RED,    /* Above high water mark */
    MTAG_METER_COUNT
} mtag_meter_levels_t;

typedef uint32_t entry_handle_t;

/* mTag table */

/**
 * @brief Add an entry to the edge identify port table
 * @param ingress_port The port number being identified
 * @param port_type The port type associated with the port
 * @param ingress_error The value to use for the error indication
 */

entry_handle_t table_identify_port_add_with_set_port_type(
    uint32_t ingress_port,
    mtag_port_type_t port_type,
    uint8_t ingress_error);

/**
 * @brief Set the default action of the identify port
 * table to send the packet to the CPU.
 * @param do_copy Set to 1 if should send copy to the CPU
 * @param cpu_code If do_copy, this is the code used
 * @param bad_packet Set to 1 to flag packet as bad
 *
 * This allows the programmer to say: If port type is not
 * set, this is an error; let me see the packet.
```

```
 *
 * Also allows just a drop of the packet.
 */

int table_indentify_port_default_common_drop_pkt(
    uint8_t do_copy,
    uint16_t cpu_code,
    uint8_t bad_packet);

/**
 * @brief Set the default action of the identify port
 * table to set to the given value
 * @param port_type The port type associated with the port
 * @param ingress_error The value to use for the error indication
 *
 * This allows the programmer to say "default port type is local"
 */

int table_indentify_port_default_common_set_port_type(
    mtag_port_type_t port_type,
    uint8_t ingress_error);

/**
 * @brief Add an entry to the add mtag table
 * @param dst_addr The L2 destination MAC for matching
 * @param vid The VLAN ID used for matching
 * @param up1 The up1 value to use in the mTag
 * @param up2 The up2 value to use in the mTag
 * @param down1 The down1 value to use in the mTag
 * @param down2 The down2 value to use in the mTag
 */
entry_handle_t table_mTag_table_add_with_add_mTag(
    mac_addr_t dst_addr, uint16_t vid,
    uint8_t up1, uint8_t up2, uint8_t down1, uint8_t down2);

/**
 * @brief Get the number of drops by ingress port and color
 * @param ingress_port The ingress port being queried.
 * @param color The color being queried.
 * @param count (output) The current value of the parameter.
 * @returns 0 on success.
 */
```

```
int counter_per_color_drops_get(
    uint32_t ingress_port,
    mtag_meter_levels_t color,
    uint64_t *count);

#endif /* MTAG_RUN_TIME_H */
```

15.8.2 Adding Hysteresis to mTag Metering with Registers

In the previous section, the mtag-edge switch used metering between local ports and the aggregation layer. Suppose that network simulation indicated a benefit if hysteresis could be used with the meters. That is, once the meter was red, packets are discarded until the meter returned to green (not just to yellow). This can be achieved by adding a register set parallel to the meters. Each cell in the register set holds the "previous" color of the meter.

Here are the changes to support this feature. The meter index is stored in local metadata for convenience.

```
////////////////////////////////////////////////////////////////
//
// headers.p4:  Add the meter index to the local metadata.
//
////////////////////////////////////////////////////////////////

header_type local_metadata_t {
    fields {
        cpu_code        : 16; // Code for packet going to CPU
        port_type       : 4;  // Type of port: up, down, local...
        ingress_error   : 1;  // An error in ingress port check
        was_mtagged     : 1;  // Track if pkt was mtagged on ingr
        copy_to_cpu     : 1;  // Special code resulting in copy to CPU
        bad_packet      : 1;  // Other error indication
        color           : 8;  // For metering
        prev_color      : 8;  // For metering hysteresis
        meter_idx       : 16; // Index used for metering
    }
}



////////////////////////////////////////////////////////////////
```

```
// mtag-edge.p4:  Declare registers and add table to update them
////////////////////////////////////////////////////////////////

// The register stores the "previous" state of the color.
// Index is the same as that used by the meter.
register prev_color {
    width : 8;
    // paired w/ meters above
    instance_count : PORT_COUNT * PORT_COUNT;
}

// Action: Update the color saved in the register
action update_prev_color(new_color) {
    modify_field(prev_color[local_metadata.meter_idx], new_color);
}

// Action: Override packet color with that from the parameter
action mark_pkt(color) {
    modify_field(local_metadata.color, color);
}

// Update meter packet action to save data
action meter_pkt(meter_idx) {
    // Save index and previous color in packet metadata
    modify_field(local_metadata.meter_idx, meter_idx);
    modify_field(local_metadata.prev_color, prev_color[meter_idx]);
    meter(per_dest_by_source, meter_idx, local_metadata.color);
}



//
// This table is statically populated with the following rules:
//    color: green,  prev_color: red    ==> update_prev_color(green)
//    color: red,    prev_color: green  ==> update_prev_color(red)
//    color: yellow, prev_color: red    ==> mark_pkt(red)
// Otherwise, no-op.
//
table hysteresis_check {
    reads {
        local_metadata.color : exact;
        local_metadata.prev_color : exact;
    }
```

```
    actions {
        update_prev_color;
        mark_pkt;
        no_op;
    }
    size : 4;
}

////////////////////////////////////////////////////////////////
// In the egress control function, check for hysteresis
////////////////////////////////////////////////////////////////

control egress {
    // Check for unknown egress state or bad retagging with mTag.
    apply(egress_check);
    apply(egress_meter) {
        hit {
            apply(hysteresis_check);
            apply(meter_policy);
        }
    }
}
```

15.8.3 ECMP Selection Example

This example shows how ECMP can be implemented using an action profile with action selector.

```
table ipv4_routing {
    reads {
        ipv4.dstAddr: lpm;
    }
    action_profile : ecmp_action_profile;
    size : 16384;    // 16K possible IPv4 prefixes
}

action_profile ecmp_action_profile {
    actions {
        nhop_set;
        no_op;
    }
```

```
    size : 4096;    // 4K possible next hops
    dynamic_action_selection : ecmp_selector;
}

// list of fields used to determine the ECMP next hop
field_list l3_hash_fields {
    ipv4.srcAddr;
    ipv4.dstAddr;
    ipv4.protocol;
    ipv4.protocol;
    tcp.sport;
    tcp.dport;
}

field_list_calculation ecmp_hash {
    input {
        l3_hash_fields;
    }
    algorithm : crc16;
    output_width : 16;
}

action_selector ecmp_selector {
    selection_key : ecmp_hash;
}
```

15.9 Feature Proposals for Future Versions

P4 is expected to evolve and develop as its features are exercised and issues are found. Incremental improvements will be released with minor version number updates. This section lists features under consideration for coming P4 versions.

Title	Summary
Support Assignment Operators	Allow fields and headers to be manipulated with assignment operators such as = or +=.
Support typing	Support data and object typing.
Better Encapsulation Support	Support better action primitives and parsing functionality for encapsulation applications.
Run Time Reconfiguration	Consider language features and conventions that would better enable consistent run time reconfigurability.
Field and Header Aliasing	Support a mechanism allowing references to different field or header instances via indirection (an alias) to allow the application of policy across multiple packet formats simultaneously.
Flexible feature inclusion	Add facilities allowing compile or run time selection of features based on availability.
Debugging Features	Support better debuggability with the addition of features such as object introspection, variable logging levels and event triggering.
Indirect Table Matching	Support database-like tables which can be queried multiple times by match+action.
Parser Repeat Loops	Support a loop construct in the parser to support variable length headers, header option lists, TLVs, etc.
Parser Select+Extract	Extend the select operator to be used in conjunction with the extract operator allowing the selection of the header instance to be extracted.

Table 9: Feature Proposals

15.10 References

[1] Bosshart, et al. *P4: Programming Protocol-Independent Packet Processors.* Computer Communication Review, July 2014. http://www.sigcomm.org/ccr/papers/2014/July/0000000.0000004.

[2] The P4 Language Consortium web site. http://www.p4.org.