P4$_{16}$ Language Specification

version 1.2.4

The P4 Language Consortium

2024-07-01

Abstract

P4 is a language for programming the data plane of network devices. This document provides a precise definition of the P4$_{16}$ language, which is the 2016 revision of the P4 language (http://p4.org). The target audience for this document includes developers who want to write compilers, simulators, IDEs, and debuggers for P4 programs. This document may also be of interest to P4 programmers who are interested in understanding the syntax and semantics of the language at a deeper level.

Contents

1. Scope

This specification document defines the structure and interpretation of programs in the $P4_{16}$ language. It defines the syntax, semantic rules, and requirements for conformant implementations of the language.

It does not define:

- Mechanisms by which P4 programs are compiled, loaded, and executed on packet-processing systems,
- Mechanisms by which data are received by one packet-processing system and delivered to another system,
- Mechanisms by which the control plane manages the match-action tables and other stateful objects defined by P4 programs,
- The size or complexity of P4 programs,

- The minimal requirements of packet-processing systems that are capable of providing a conformant implementation.

It is understood that some implementations may be unable to implement the behavior defined here in all cases, or may provide options to eliminate some safety guarantees in exchange for better performance or handling larger programs. They should document where they deviate from this specification.

2. Terms, definitions, and symbols

Throughout this document, the following terms will be used:

- Architecture: A set of P4-programmable components and the data plane interfaces between them.
- Control plane: A class of algorithms and the corresponding input and output data that are concerned with the provisioning and configuration of the data plane.
- Data plane: A class of algorithms that describe transformations on packets by packet-processing systems.
- Metadata: Intermediate data generated during execution of a P4 program.
- Packet: A network packet is a formatted unit of data carried by a packet-switched network.
- Packet header: Formatted data at the beginning of a packet. A given packet may contain a sequence of packet headers representing different network protocols.
- Packet payload: Packet data that follows the packet headers.
- Packet-processing system: A data-processing system designed for processing network packets. In general, packet-processing systems implement control plane and data plane algorithms.
- Target: A packet-processing system capable of executing a P4 program.

All terms defined explicitly in this document should not be understood to refer implicitly to similar terms defined elsewhere. Conversely, any terms not defined explicitly in this document should be interpreted according to generally recognizable sources—e.g., IETF RFCs.

3. Overview

P4 is a language for expressing how packets are processed by the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance. The name P4 comes from the original paper that introduced the language, "Programming Protocol-independent Packet Processors," https://arxiv.org/pdf/1312.1719.pdf. While P4 was initially designed for programming switches, its scope has been broadened to cover a large variety of devices. In the rest of this document we use the generic term target for all such devices.

Many targets implement both a control plane and a data plane. P4 is designed to specify only the data plane functionality of the target. P4 programs also partially define the interface by which the control plane and the data-plane communicate, but P4 cannot be used to describe the control-plane functionality of the target. In the rest of this document, when we talk about P4 as "programming a target", we mean "programming the data plane of a target".

As a concrete example of a target, Figure 1 illustrates the difference between a traditional fixed-function switch and a P4-programmable switch. In a traditional switch the manufacturer defines the data-plane functionality. The control-plane controls the data plane by managing entries in tables (e.g.

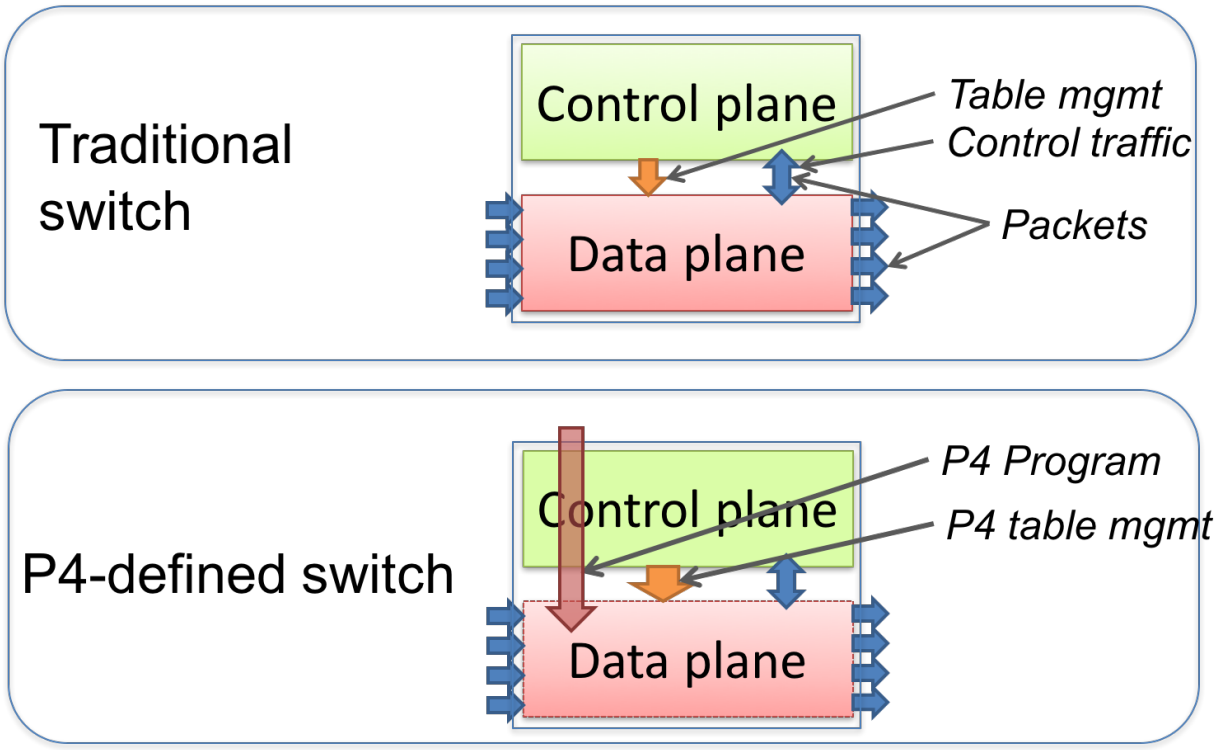

Figure 1. Traditional switches vs. programmable switches.

routing tables), configuring specialized objects (e.g. meters), and by processing control-packets (e.g. routing protocol packets) or asynchronous events, such as link state changes or learning notifications.

A P4-programmable switch differs from a traditional switch in two essential ways:

- The data plane functionality is not fixed in advance but is defined by a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program (shown by the long red arrow) and has no built-in knowledge of existing network protocols.
- The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

Hence, P4 can be said to be protocol independent, but it enables programmers to express a rich set of protocols and other data plane behaviors.

The core abstractions provided by the P4 language are:

- Header types describe the format (the set of fields and their sizes) of each header within a packet.
- Parsers describe the permitted sequences of headers within received packets, how to identify those header sequences, and the headers and fields to extract from packets.
- Tables associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions.
- Actions are code fragments that describe how packet header fields and metadata are manipu-

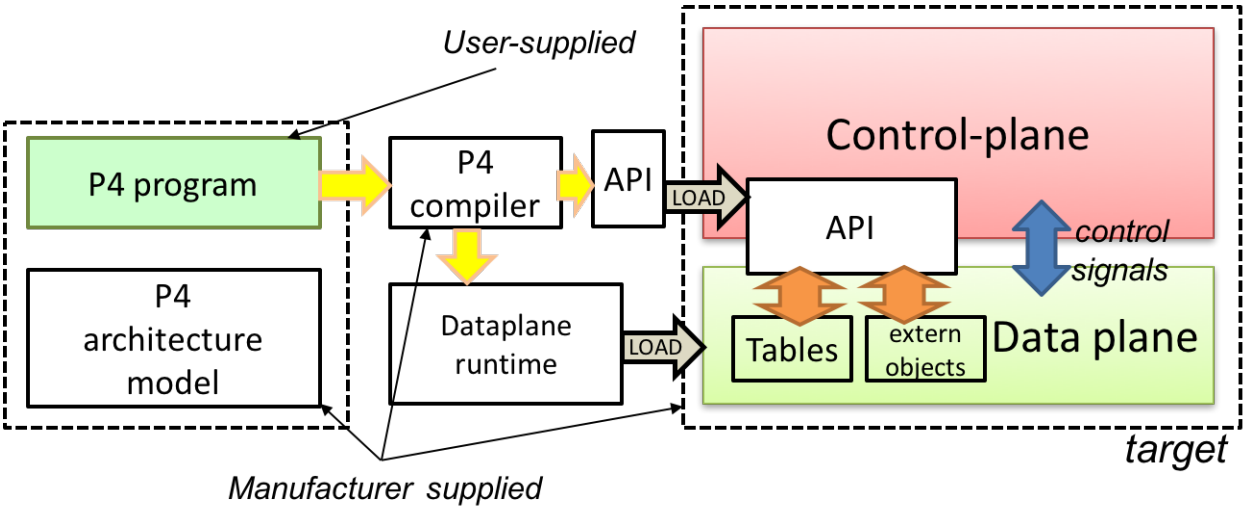

Figure 2. Programming a target with P4.

lated. Actions can include data, which is supplied by the control-plane at runtime.

- Match-action units perform the following sequence of operations:
  - Construct lookup keys from packet fields or computed metadata,
  - Perform table lookup using the constructed key, choosing an action (including the associated data) to execute, and
  - Finally, execute the selected action.
- Control flow expresses an imperative program that describes packet-processing on a target, including the data-dependent sequence of match-action unit invocations. Deparsing (packet reassembly) can also be performed using a control flow.
- Extern objects are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g., checksum units) and hence not programmable using P4.
- User-defined metadata: user-defined data structures associated with each packet.
- Intrinsic metadata: metadata provided by the architecture associated with each packet—e.g., the input port where a packet has been received.

Figure 2 shows a typical tool workflow when programming a target using P4.

Target manufacturers provide the hardware or software implementation framework, an architecture definition, and a P4 compiler for that target. P4 programmers write programs for a specific architecture, which defines a set of P4-programmable components on the target as well as their external data plane interfaces.

Compiling a set of P4 programs produces two artifacts:

- a data plane configuration that implements the forwarding logic described in the input program and
- an API for managing the state of the data plane objects from the control plane

P4 is a domain-specific language that is designed to be implementable on a large variety of targets in-

cluding programmable network interface cards, FPGAs, software switches, and hardware ASICs. As such, the language is restricted to constructs that can be efficiently implemented on all of these platforms.

Assuming a fixed cost for table lookup operations and interactions with extern objects, all P4 programs (i.e., parsers and controls) execute a constant number of operations for each byte of an input packet received and analyzed. Although parsers may contain loops, provided some header is extracted on each cycle, the packet itself provides a bound on the total execution of the parser. In other words, under these assumptions, the computational complexity of a P4 program is linear in the total size of all headers, and never depends on the size of the state accumulated while processing data (e.g., the number of flows, or the total number of packets processed). These guarantees are necessary (but not sufficient) for enabling fast packet processing across a variety of targets.

P4 conformance of a target is defined as follows: if a specific target T supports only a subset of the P4 programming language, say $P4^{T}$, programs written in $P4^{T}$ executed on the target should provide the exact same behavior as is described in this document. Note that P4 conformant targets can provide arbitrary P4 language extensions and `extern` elements.

3.1. Benefits of P4

Compared to state-of-the-art packet-processing systems (e.g., based on writing microcode on top of custom hardware), P4 provides a number of significant advantages:

- Flexibility: P4 makes many packet-forwarding policies expressible as programs, in contrast to traditional switches, which expose fixed-function forwarding engines to their users.
- Expressiveness: P4 can express sophisticated, hardware-independent packet processing algorithms using solely general-purpose operations and table look-ups. Such programs are portable across hardware targets that implement the same architectures (assuming sufficient resources are available).
- Resource mapping and management: P4 programs describe storage resources abstractly (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage low-level details such as allocation and scheduling.
- Software engineering: P4 programs provide important benefits such as type checking, information hiding, and software reuse.
- Component libraries: Component libraries supplied by manufacturers can be used to wrap hardware-specific functions into portable high-level P4 constructs.
- Decoupling hardware and software evolution: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing.
- Debugging: Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.

3.2. P4 language evolution: comparison to previous versions (P4 v1.0/v1.1)

Compared to $P4_{14}$, the earlier version of the language, $P4_{16}$ makes a number of significant, backwards-incompatible changes to the syntax and semantics of the language. The evolution from the previous version ($P4_{14}$) to the current one ($P4_{16}$) is depicted in Figure 3. In particular, a large number of language features have been eliminated from the language and moved into libraries including counters, checksum units, meters, etc.

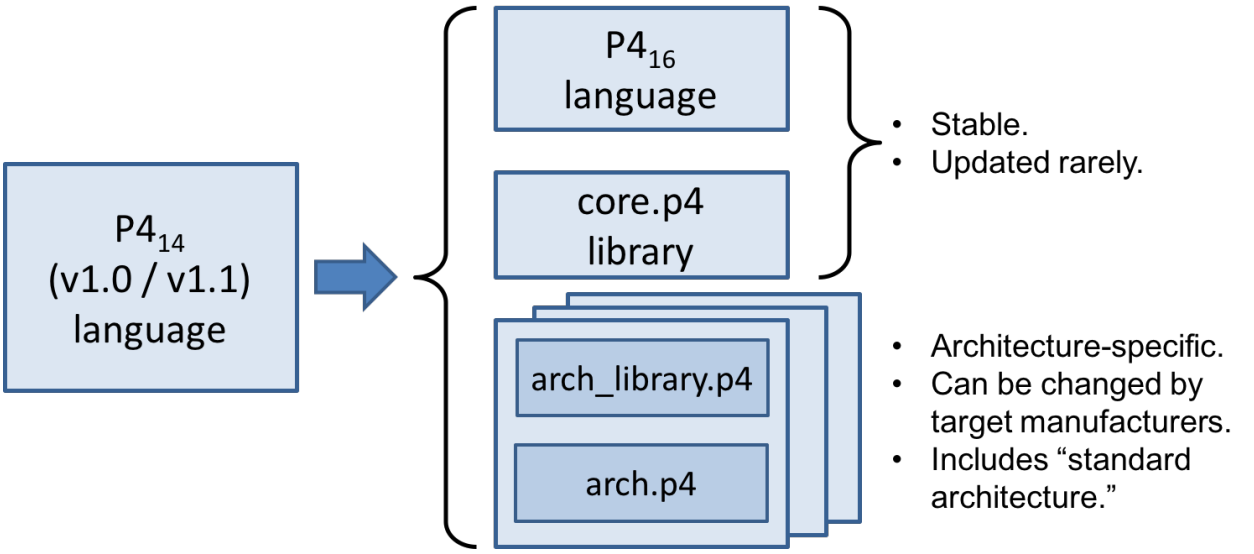

Figure 3. Evolution of the language between versions $P4_{14}$ (versions 1.0 and 1.1) and $P4_{16}$.

Hence, the language has been transformed from a complex language (more than 70 keywords) into a relatively small core language (less than 40 keywords, shown in Section B) accompanied by a library of fundamental constructs that are needed for writing most P4.

The v1.1 version of P4 introduced a language construct called `extern` that can be used to describe library elements. Many constructs defined in the v1.1 language specification will thus be transformed into such library elements (including constructs that have been eliminated from the language, such as counters and meters). Some of these `extern` objects are expected to be standardized, and they will be in the scope of a future document describing a standard library of P4 elements. In this document we provide several examples of `extern` constructs. $P4_{16}$ also introduces and repurposes some v1.1 language constructs for describing the programmable parts of an architecture. These language constructs are: `parser`, `state`, `control`, and `package`.

One important goal of the $P4_{16}$ language revision is to provide a stable language definition. In other words, we strive to ensure that all programs written in $P4_{16}$ will remain syntactically correct and behave identically when treated as programs for future versions of the language. Moreover, if some future version of the language requires breaking backwards compatibility, we will seek to provide an easy path for migrating $P4_{16}$ programs to the new version.

4. Architecture Model

The P4 architecture identifies the P4-programmable blocks (e.g., parser, ingress control flow, egress control flow, deparser, etc.) and their data plane interfaces.

The P4 architecture can be thought of as a contract between the program and the target. Each manufacturer must therefore provide both a P4 compiler as well as an accompanying architecture definition for their target. (We expect that P4 compilers can share a common front-end that handles all architectures). The architecture definition does not have to expose the entire programmable surface of the data plane—a manufacturer may even choose to provide multiple definitions for the same hardware device, each with different capabilities (e.g., with or without multicast support).

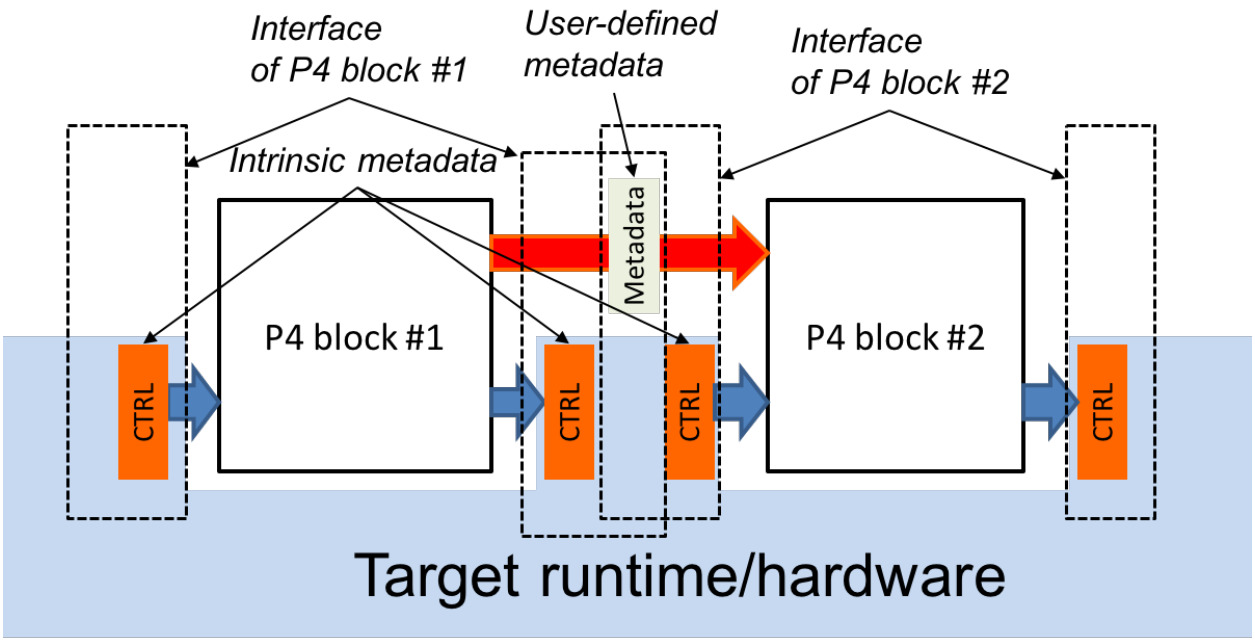

Figure 4. P4 program interfaces.

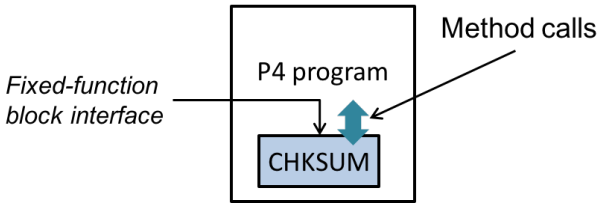

Figure 5. P4 program invoking the services of a fixed-function object.

Figure 4 illustrates the data plane interfaces between P4-programmable blocks. It shows a target that has two programmable blocks (#1 and #2). Each block is programmed through a separate fragment of P4 code. The target interfaces with the P4 program through a set of control registers or signals. Input controls provide information to P4 programs (e.g., the input port that a packet was received from), while output controls can be written to by P4 programs to influence the target behavior (e.g., the output port where a packet has to be directed). Control registers/signals are represented in P4 as intrinsic metadata. P4 programs can also store and manipulate data pertaining to each packet as user-defined metadata.

The behavior of a P4 program can be fully described in terms of transformations that map vectors of bits to vectors of bits. To actually process a packet, the architecture model interprets the bits that the P4 program writes to intrinsic metadata. For example, to cause a packet to be forwarded on a specific output port, a P4 program may need to write the index of an output port into a dedicated control register. Similarly, to cause a packet to be dropped, a P4 program may need to set a "drop" bit into another dedicated control register. Note that the details of how intrinsic metadata are interpreted is architecture-specific.

P4 programs can invoke services implemented by extern objects and functions provided by the

architecture. Figure 5 depicts a P4 program invoking the services of a built-in checksum computation unit on a target. The implementation of the checksum unit is not specified in P4, but its interface is. In general, the interface for an extern object describes each operation it provides, as well as their parameter and return types.

In general, P4 programs are not expected to be portable across different architectures. For example, executing a P4 program that broadcasts packets by writing into a custom control register will not function correctly on a target that does not have the control register. However, P4 programs written for a given architecture should be portable across all targets that faithfully implement the corresponding model, provided there are sufficient resources.

4.1. Standard architectures

We expect that the P4 community will evolve a small set of standard architecture models pertaining to specific verticals. Wide adoption of such standard architectures will promote portability of P4 programs across different targets. However, defining these standard architectures is outside of the scope of this document.

4.2. Data plane interfaces

To describe a functional block that can be programmed in P4, the architecture includes a type declaration that specifies the interfaces between the block and the other components in the architecture. For example, the architecture might contain a declaration such as the following:

```
control MatchActionPipe<H>(in bit<4> inputPort,
                           inout H parsedHeaders,
                           out bit<4> outputPort);
```

This type declaration describes a block named `MatchActionPipe` that can be programmed using a data-dependent sequence of match-action unit invocations and other imperative constructs (indicated by the **`control`** keyword). The interface between the `MatchActionPipe` block and the other components of the architecture can be read off from this declaration:

- The first parameter is a 4-bit value named `inputPort`. The direction **`in`** indicates that this parameter is an input that cannot be modified.
- The second parameter is an object of type `H` named `parsedHeaders`, where `H` is a type variable representing the headers that will be defined later by the P4 programmer. The direction **`inout`** indicates that this parameter is both an input and an output.
- The third parameter is a 4-bit value named `outputPort`. The direction **`out`** indicates that this parameter is an output whose value is undefined initially but can be modified.

4.3. Extern objects and functions

P4 programs can also interact with objects and functions provided by the architecture. Such objects are described using the **`extern`** construct, which describes the interfaces that such objects expose to the data-plane.

An **`extern`** object describes a set of methods that are implemented by an object, but not the implementation of these methods (i.e., it is similar to an abstract class in an object-oriented language). For

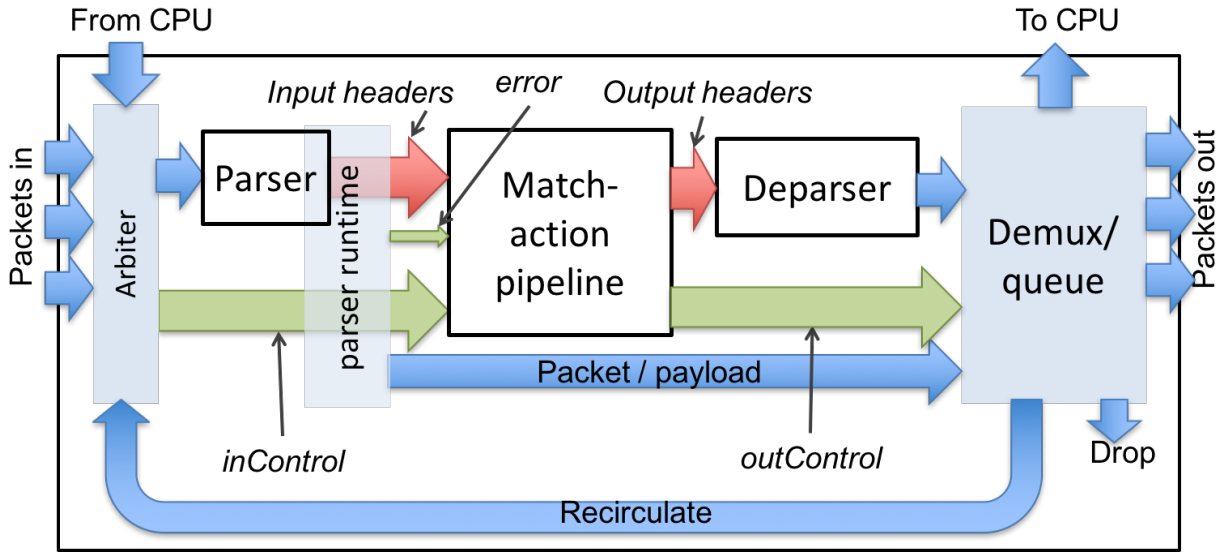

Figure 6. The Very Simple Switch (VSS) architecture.

example, the following construct could be used to describe the operations offered by an incremental checksum unit:

```
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

5. Example: A very simple switch

As an example to illustrate the features of architectures, consider implementing a very simple switch in P4. We will first describe the architecture of the switch and then write a complete P4 program that specifies the data plane behavior of the switch. This example demonstrates many important features of the P4 programming language.

We call our architecture the "Very Simple Switch" (VSS). Figure 6 is a diagram of this architecture. There is nothing inherently special about VSS—it is just a pedagogical example that illustrates how programmable switches can be described and programmed in P4. VSS has a number of fixed-function blocks (shown in cyan in our example), whose behavior is described in Section 5.2. The white blocks are programmable using P4.

VSS receives packets through one of 8 input Ethernet ports, through a recirculation channel, or from a port connected directly to the CPU. VSS has one single parser, feeding into a single match-action pipeline, which feeds into a single deparser. After exiting the deparser, packets are emitted through one of 8 output Ethernet ports or one of 3 "special" ports:

- Packets sent to the “CPU port” are sent to the control plane
- Packets sent to the “Drop port” are discarded
- Packets sent to the “Recirculate port” are re-injected in the switch through a special input port

The white blocks in the figure are programmable, and the user must provide a corresponding P4 program to specify the behavior of each such block. The red arrows indicate the flow of user-defined data. The cyan blocks are fixed-function components. The green arrows are data plane interfaces used to convey information between the fixed-function blocks and the programmable blocks—exposed in the P4 program as intrinsic metadata.

5.1. Very Simple Switch Architecture

The following P4 program provides a declaration of VSS in P4, as it would be provided by the VSS manufacturer. The declaration contains several type declarations, constants, and finally declarations for the three programmable blocks; the code uses syntax highlighting. The programmable blocks are described by their types; the implementation of these blocks has to be provided by the switch programmer.

```
// File "very_simple_switch_model.p4"
// Very Simple Switch P4 declaration
// core library needed for packet_in and packet_out definitions
# include <core.p4>
/* Various constants and structure declarations */
/* ports are represented using 4-bit values */
typedef bit<4> PortId;
/* only 8 ports are "real" */
const PortId REAL_PORT_COUNT = 4w8;  // 4w8 is the number 8 in 4 bits
/* metadata accompanying an input packet */
struct InControl {
    PortId inputPort;
}
/* special input port values */
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;
/* metadata that must be computed for outgoing packets */
struct OutControl {
    PortId outputPort;
}
/* special output port values for outgoing packet */
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;
/* Prototypes for all programmable blocks */
/**
 * Programmable parser.
 * @param <H> type of headers; defined by user
 * @param b input packet
```

```
 * @param parsedHeaders headers constructed by parser
 */
parser Parser<H>(packet_in b,
                 out H parsedHeaders);
/**
 * Match-action pipeline
 * @param <H> type of input and output headers
 * @param headers headers received from the parser and sent to the deparser
 * @param parseError error that may have surfaced during parsing
 * @param inCtrl information from architecture, accompanying input packet
 * @param outCtrl information for architecture, accompanying output packet
 */
control Pipe<H>(inout H headers,
                in error parseError,// parser error
                in InControl inCtrl,// input port
                out OutControl outCtrl); // output port
/**
 * VSS deparser.
 * @param <H> type of headers; defined by user
 * @param b output packet
 * @param outputHeaders headers for output packet
 */
control Deparser<H>(inout H outputHeaders,
                    packet_out b);
/**
 * Top-level package declaration - must be instantiated by user.
 * The arguments to the package indicate blocks that
 * must be instantiated by the user.
 * @param <H> user-defined type of the headers processed.
 */
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
// Architecture-specific objects that can be instantiated
// Checksum unit
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

Let us describe some of these elements:

- The included file `core.p4` is described in more detail in Appendix D. It defines some standard data-types and error codes.

- `bit<4>` is the type of bit-strings with 4 bits.
- The syntax `4w0xF` indicates the value 15 represented using 4 bits. An alternative notation is `4w15`. In many circumstances the width modifier can be omitted, writing just `15`.
- `error` is a built-in P4 type for holding error codes
- Next follows the declaration of a parser:

```
parser Parser<H>(packet_in b, out H parsedHeaders);
```

  This declaration describes the interface for a parser, but not yet its implementation, which will be provided by the programmer. The parser reads its input from a `packet_in`, which is a pre-defined P4 extern object that represents an incoming packet, declared in the `core.p4` library. The parser writes its output (the `out` keyword) into the `parsedHeaders` argument. The type of this argument is `H`, yet unknown—it will also be provided by the programmer.

- The declaration

```
control Pipe<H>(inout H headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl);
```

  describes the interface of a Match-Action pipeline named `Pipe`.

The pipeline receives three inputs: the headers `headers`, a parser error `parseError`, and the `inCtrl` control data. Figure 6 indicates the different sources of these pieces of information. The pipeline writes its outputs into `outCtrl`, and it must update in place the headers to be consumed by the deparser.

- The top-level package is called `VSS`; in order to program a VSS, the user will have to instantiate a package of this type (shown in the next section). The top-level package declaration also depends on a type variable H:

```
package VSS<H>
```

A type variable indicates a type yet unknown that must be provided by the user at a later time. In this case `H` is the type of the set of headers that the user program will be processing; the parser will produce the parsed representation of these headers, and the match-action pipeline will update the input headers in place to produce the output headers.

- The `package VSS` declaration has three complex parameters, of types `Parser`, `Pipe`, and `Deparser` respectively; which are exactly the declarations we have just described. In order to program the target one has to supply values for these parameters.
- In this program the `inCtrl` and `outCtrl` structures represent control registers. The content of the headers structure is stored in general-purpose registers.

- The `extern Checksum16` declaration describes an extern object whose services can be invoked to compute checksums.

5.2. Very Simple Switch Architecture Description

In order to fully understand VSS's behavior and write meaningful P4 programs for it, and for implementing a control plane, we also need a full behavioral description of the fixed-function blocks. This section can be seen as a simple example illustrating all the details that have to be handled when writing an architecture description. The P4 language is not intended to cover the description of all such functional blocks—the language can only describe the interfaces between programmable blocks and the architecture. For the current program, this interface is given by the `Parser`, `Pipe`, and `Deparser` declarations. In practice we expect that the complete description of the architecture will be provided as an executable program and/or diagrams and text; in this document we will provide informal descriptions in English.

5.2.1. Arbiter block

The input arbiter block performs the following functions:

- It receives packets from one of the physical input Ethernet ports, from the control plane, or from the input recirculation port.
- For packets received from Ethernet ports, the block computes the Ethernet trailer checksum and verifies it. If the checksum does not match, the packet is discarded. If the checksum does match, it is removed from the packet payload.
- Receiving a packet involves running an arbitration algorithm if multiple packets are available.
- If the arbiter block is busy processing a previous packet and no queue space is available, input ports may drop arriving packets, without indicating the fact that the packets were dropped in any way.
- After receiving a packet, the arbiter block sets the `inCtrl.inputPort` value that is an input to the match-action pipeline with the identity of the input port where the packet originated. Physical Ethernet ports are numbered 0 to 7, while the input recirculation port has a number 13 and the CPU port has the number 14.

5.2.2. Parser runtime block

The parser runtime block works in concert with the parser. It provides an error code to the match-action pipeline, based on the parser actions, and it provides information about the packet payload (e.g., the size of the remaining payload data) to the demux block. As soon as a packet's processing is completed by the parser, the match-action pipeline is invoked with the associated metadata as inputs (packet headers and user-defined metadata).

5.2.3. Demux block

The core functionality of the demux block is to receive the headers for the outgoing packet from the deparser and the packet payload from the parser, to assemble them into a new packet and to send the result to the correct output port. The output port is specified by the value of `outCtrl.ouputPort`, which is set by the match-action pipeline.

- Sending the packet to the drop port causes the packet to disappear.
- Sending the packet to an output Ethernet port numbered between 0 and 7 causes it to be emitted on the corresponding physical interface. The packet may be placed in a queue if the output interface is already busy emitting another packet. When the packet is emitted, the physical interface computes a correct Ethernet checksum trailer and appends it to the packet.
- Sending a packet to the output CPU port causes the packet to be transferred to the control plane. In this case, the packet that is sent to the CPU is the original input packet, and not the packet received from the deparser—the latter packet is discarded.
- Sending the packet to the output recirculation port causes it to appear at the input recirculation port. Recirculation is useful when packet processing cannot be completed in a single pass.
- If the `outputPort` has an illegal value (e.g., 9), the packet is dropped.
- Finally, if the demux unit is busy processing a previous packet and there is no capacity to queue the packet coming from the deparser, the demux unit may drop the packet, irrespective of the output port indicated.

Please note that some of the behaviors of the demux block may be unexpected—we have highlighted them in bold. We are not specifying here several important behaviors related to queue size, arbitration, and timing, which also influence the packet processing.

The arrow shown from the parser runtime to the demux block represents an additional information flow from the parser to the demux: the packet being processed as well as the offset within the packet where parsing ended (i.e., the start of the packet payload).

5.2.4. Available extern blocks

The VSS architecture provides an incremental checksum extern block, called `Checksum16`. The checksum unit has a constructor and four methods:

- `clear()`: prepares the unit for a new computation
- `update<T>(in T data)`: add some data to be checksummed. The data must be either a bit-string, a header-typed value, or a **`struct`** containing such values. The fields in the header/struct are concatenated in the order they appear in the type declaration.
- `get()`: returns the 16-bit one's complement checksum. When this function is invoked the checksum must have received an integral number of bytes of data.
- `remove<T>(in T data)`: assuming that `data` was used for computing the current checksum, `data` is removed from the checksum.

5.3. A complete Very Simple Switch program

Here we provide a complete P4 program that implements basic forwarding for IPv4 packets on the VSS architecture. This program does not utilize all of the features provided by the architecture—e.g., recirculation—but it does use preprocessor `#include` directives (see Section 6.2).

The parser attempts to recognize an Ethernet header followed by an IPv4 header. If either of these headers are missing, parsing terminates with an error. Otherwise it extracts the information from these headers into a `Parsed_packet` structure. The match-action pipeline is shown in Figure 7; it comprises four match-action units (represented by the P4 **`table`** keyword):

- If any parser error has occurred, the packet is dropped (i.e., by assigning `outputPort` to `DROP_PORT`)

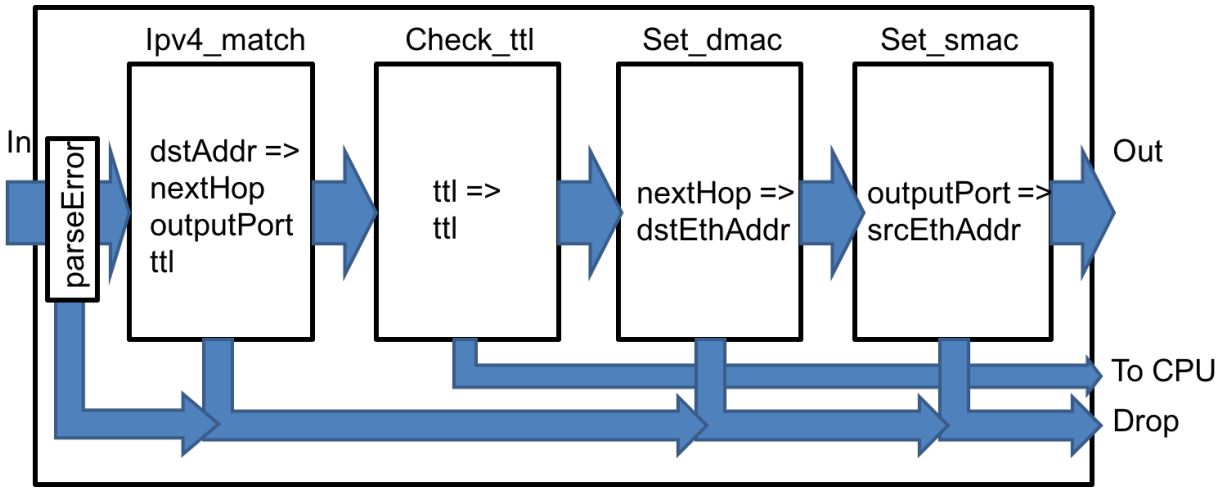

Figure 7. Diagram of the match-action pipeline expressed by the VSS P4 program.

- The first table uses the IPv4 destination address to determine the `outputPort` and the IPv4 address of the next hop. If this lookup fails, the packet is dropped. The table also decrements the IPv4 `ttl` value.
- The second table checks the `ttl` value: if the `ttl` becomes 0, the packet is sent to the control plane through the CPU port.
- The third table uses the IPv4 address of the next hop (which was computed by the first table) to determine the Ethernet address of the next hop.
- Finally, the last table uses the `outputPort` to identify the source Ethernet address of the current switch, which is set in the outgoing packet.

The deparser constructs the outgoing packet by reassembling the Ethernet and IPv4 headers as computed by the pipeline.

```
// Include P4 core library
# include <core.p4>

// Include very simple switch architecture declarations
# include "very_simple_switch_model.p4"

// This program processes packets comprising an Ethernet and an IPv4
// header, and it forwards packets using the destination IP address

typedef bit<48>  EthernetAddress;
typedef bit<32>  IPv4Address;

// Standard Ethernet header
header Ethernet_h {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
```

```
    bit<16>         etherType;
}

// IPv4 header (without options)
header IPv4_h {
    bit<4>       version;
    bit<4>       ihl;
    bit<8>       diffserv;
    bit<16>      totalLen;
    bit<16>      identification;
    bit<3>       flags;
    bit<13>      fragOffset;
    bit<8>       ttl;
    bit<8>       protocol;
    bit<16>      hdrChecksum;
    IPv4Address  srcAddr;
    IPv4Address  dstAddr;
}

// Structure of parsed headers
struct Parsed_packet {
    Ethernet_h ethernet;
    IPv4_h     ip;
}

// Parser section

// User-defined errors that may be signaled during parsing
error {
    IPv4OptionsNotSupported,
    IPv4IncorrectVersion,
    IPv4ChecksumError
}

parser TopParser(packet_in b, out Parsed_packet p) {
    Checksum16() ck;  // instantiate checksum unit

    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
            // no default rule: all other packets rejected
        }
    }

    state parse_ipv4 {
```

```
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
        ck.clear();
        ck.update(p.ip);
        // Verify that packet checksum is zero
        verify(ck.get() == 16w0, error.IPv4ChecksumError);
        transition accept;
    }
}

// Match-action pipeline section

control TopPipe(inout Parsed_packet headers,
                in error parseError, // parser error
                in InControl inCtrl, // input port
                out OutControl outCtrl) {
    IPv4Address nextHop;  // local variable

    /**
     * Indicates that a packet is dropped by setting the
     * output port to the DROP_PORT
     */
     action Drop_action() {
         outCtrl.outputPort = DROP_PORT;
     }

    /**
     * Set the next hop and the output port.
     * Decrements ipv4 ttl field.
     * @param ipv4_dest ipv4 address of next hop
     * @param port output port
     */
     action Set_nhop(IPv4Address ipv4_dest, PortId port) {
         nextHop = ipv4_dest;
         headers.ip.ttl = headers.ip.ttl - 1;
         outCtrl.outputPort = port;
     }

    /**
     * Computes address of next IPv4 hop and output port
     * based on the IPv4 destination of the current packet.
     * Decrements packet IPv4 TTL.
     * @param nextHop IPv4 address of next hop
     */
    table ipv4_match {
```

```
        key = { headers.ip.dstAddr: lpm; }  // longest-prefix match
        actions = {
             Drop_action;
             Set_nhop;
        }
        size = 1024;
        default_action = Drop_action;
    }

    /**
     * Send the packet to the CPU port
     */
     action Send_to_cpu() {
         outCtrl.outputPort = CPU_OUT_PORT;
     }

    /**
     * Check packet TTL and send to CPU if expired.
     */
    table check_ttl {
        key = { headers.ip.ttl: exact; }
        actions = { Send_to_cpu; NoAction; }
        const default_action = NoAction; // defined in core.p4
    }

    /**
     * Set the destination MAC address of the packet
     * @param dmac destination MAC address.
     */
     action Set_dmac(EthernetAddress dmac) {
         headers.ethernet.dstAddr = dmac;
     }

    /**
     * Set the destination Ethernet address of the packet
     * based on the next hop IP address.
     * @param nextHop IPv4 address of next hop.
     */
     table dmac {
         key = { nextHop: exact; }
         actions = {
              Drop_action;
              Set_dmac;
         }
         size = 1024;
         default_action = Drop_action;
```

```
    }

    /**
     * Set the source MAC address.
     * @param smac: source MAC address to use
     */
     action Set_smac(EthernetAddress smac) {
         headers.ethernet.srcAddr = smac;
     }

    /**
     * Set the source mac address based on the output port.
     */
    table smac {
         key = { outCtrl.outputPort: exact; }
         actions = {
              Drop_action;
              Set_smac;
        }
        size = 16;
        default_action = Drop_action;
    }

    apply {
        if (parseError != error.NoError) {
            Drop_action();  // invoke drop directly
            return;
        }

        ipv4_match.apply(); // Match result will go into nextHop
        if (outCtrl.outputPort == DROP_PORT) return;

        check_ttl.apply();
        if (outCtrl.outputPort == CPU_OUT_PORT) return;

        dmac.apply();
        if (outCtrl.outputPort == DROP_PORT) return;

        smac.apply();
    }
}

// deparser section
control TopDeparser(inout Parsed_packet p, packet_out b) {
    Checksum16() ck;
    apply {
```

```
        b.emit(p.ethernet);
        if (p.ip.isValid()) {
            ck.clear();              // prepare checksum unit
            p.ip.hdrChecksum = 16w0; // clear checksum
            ck.update(p.ip);         // compute new checksum.
            p.ip.hdrChecksum = ck.get();
        }
        b.emit(p.ip);
    }
}

// Instantiate the top-level VSS package
VSS(TopParser(),
    TopPipe(),
    TopDeparser()) main;
```

6. P4 language definition

The P4 language can be viewed as having several distinct components, which we describe separately:

- The core language, comprising of types, variables, scoping, declarations, statements, expressions, etc. We start by describing this part of the language.
- A sub-language for expressing parsers, based on state machines (Section 13).
- A sub-language for expressing computations using match-action units, based on traditional imperative control-flow (Section 14).
- A sub-language for describing architectures (Section 17).

6.1. Syntax and semantics

6.1.1. Grammar

The complete grammar of $P4_{16}$ is given in Appendix G, using Yacc/Bison grammar description language. This text is based on the same grammar. We adopt several standard conventions when we provide excerpts from the grammar:

- UPPERCASE symbols denote terminals in the grammar.
- Excerpts from the grammar are given in BNF notation as follows:

```
p4program
  : /* empty */
  | p4program declaration
  | p4program ";"  /* empty declaration */
  ;
```

Pseudo-code (mostly used for describing the semantics of various P4 constructs) are shown with fixed-size fonts as in the following example:

```
ParserModel.verify(bool condition, error err) {
    if (condition == false) {
        ParserModel.parserError = err;
        goto reject;
    }
}
```

6.1.2. Semantics and the P4 abstract machines

We describe the semantics of P4 in terms of abstract machines executing traditional imperative code. There is an abstract machine for each P4 sub-language (parser, control). The abstract machines are described in this text in pseudo-code and English.

P4 compilers are free to reorganize the code they generate in any way as long as the externally visible behaviors of the P4 programs are preserved as described by this specification where externally visible behavior is defined as:

- The input/output behavior of all P4 blocks, and
- The state maintained by extern blocks.

6.2. Preprocessing

To aid composition of programs from multiple source files P4 compilers should support the following subset of the C preprocessor functionality:

- `#define` for defining macros (without arguments)
- `#undef`
- `#if #else #endif #ifdef #ifndef #elif`
- `#include`

The preprocessor should also remove the sequence backslash newline (ASCII codes 92, 10) to facilitate splitting content across multiple lines when convenient for formatting.

Additional C preprocessor capabilities may be supported, but are not guaranteed—e.g., macros with arguments. Similar to C, `#include` can specify a file name either within double quotes or within <>.

```
# include <system_file>
# include "user_file"
```

The difference between the two forms is the order in which the preprocessor searches for header files when the path is incompletely specified.

P4 compilers should correctly handle `#line` directives that may be generated during preprocessing. This functionality allows P4 programs to be built from multiple source files, potentially produced by different programmers at different times:

- the P4 core library, defined in this document,
- the architecture, defining data plane interfaces and extern blocks,
- user-defined libraries of useful components (e.g. standard protocol header definitions), and
- the P4 programs that specify the behavior of each programmable block.

6.3. P4 core library

The P4 language specification defines a core library that includes several common programming constructs. A description of the core library is provided in Appendix D. All P4 programs must include the core library. Including the core library is done with

```
# include <core.p4>
```

6.4. Lexical constructs

All P4 keywords use only ASCII characters. All P4 identifiers must use only ASCII characters. P4 compilers should handle correctly strings containing 8-bit characters in comments and string literals. P4 is case-sensitive. Whitespace characters, including newlines are treated as token separators. Indentation is free-form; however, P4 has C-like block constructs, and all our examples use C-style indentation. Tab characters are treated as spaces.

The lexer recognizes the following kinds of terminals:

- `IDENTIFIER`: start with a letter or underscore, and contain letters, digits and underscores
- `TYPE_IDENTIFIER`: identifier that denotes a type name
- `INTEGER`: integer literals
- `DONTCARE`: a single underscore
- Keywords such as `RETURN`. By convention, each keyword terminal corresponds to a language keyword with the same spelling but using lowercase. For example, the `RETURN` terminal corresponds to the **`return`** keyword.

6.4.1. Identifiers

P4 identifiers may contain only letters, numbers, and the underscore character _, and must start with a letter or underscore. The special identifier consisting of a single underscore _ is reserved to indicate a “don't care” value; its type may vary depending on the context. Certain keywords (e.g., **`apply`**) can be used as identifiers if the context makes it unambiguous.

```
nonTypeName
    : IDENTIFIER
    | APPLY
    | KEY
    | ACTIONS
    | STATE
    | ENTRIES
    | TYPE
    | PRIORITY
    ;

name
    : nonTypeName
    | TYPE_IDENTIFIER
```

```
    ;
```

6.4.2. Comments

P4 supports several kinds of comments:

- Single-line comments, introduced by `//` and spanning to the end of line,
- Multi-line comments, enclosed between `/*` and `*/`
- Nested multi-line comments are not supported.
- Javadoc-style comments, starting with `/**` and ending with `*/`

Use of Javadoc-style comments is strongly encouraged for the tables and actions that are used to synthesize the interface with the control-plane.

P4 treats comments as token separators and no comments are allowed within a token—e.g. `bi/**/t` is parsed as two tokens, `bi` and `t`, and not as a single token **`bit`**.

6.4.3. Literal constants

6.4.3.1. Boolean literals There are two Boolean literal constants: **`true`** and **`false`**.

6.4.3.2. Integer literals Integer literals are non-negative arbitrary-precision integers. By default, literals are represented in base 10. The following prefixes must be employed to specify the base explicitly:

- `0x` or `0X` indicates base 16 (hexadecimal)
- `0o` or `0O` indicates base 8 (octal)
- `0d` or `0D` indicates base 10 (decimal)
- `0b` or `0B` indicates base 2

The width of a numeric literal in bits can be specified by an unsigned number prefix consisting of a number of bits and a signedness indicator:

- `w` indicates unsigned numbers
- `s` indicates signed numbers

Note that a leading zero by itself does not indicate an octal (base 8) constant. The underscore character is considered a digit within number literals but is ignored when computing the value of the parsed number. This allows long constant numbers to be more easily read by grouping digits together. The underscore cannot be used in the width specification or as the first character of an integer literal. No comments or whitespaces are allowed within a literal. Here are some examples of numeric literals:

```
32w255         // a 32-bit unsigned number with value 255
32w0d255       // same value as above
32w0xFF        // same value as above
32s0xFF        // a 32-bit signed number with value 255
8w0b10101010   // an 8-bit unsigned number with value 0xAA
8w0b_1010_1010 // same value as above
8w170          // same value as above
```

```
8s0b1010_1010  // an 8-bit signed number with value -86
16w0377        // 16-bit unsigned number with value 377 (not 255!)
16w0o377       // 16-bit unsigned number with value 255 (base 8)
```

6.4.3.3. String literals

String literals are specified as an arbitrary sequence of 8-bit characters, enclosed within double quote characters " (ASCII code 34). Strings start with a double quote character and extend to the first double quote sign which is not immediately preceded by an odd number of backslash characters (ASCII code 92). P4 does not make any validity checks on strings (i.e., it does not check that strings represent legal UTF-8 encodings).

Since P4 does not provide any operations on strings, string literals are generally passed unchanged through the P4 compiler to other third-party tools or compiler-backends, including the terminating quotes. These tools can define their own handling of escape sequences (e.g., how to specify Unicode characters, or handle unprintable ASCII characters).

Here are 3 examples of string literals:

```
"simple string"
"string \" with \" embedded \" quotes"
"string with embedded
line terminator"
```

6.4.4. Optional trailing commas

The P4 grammar allows several kinds of comma-separated lists to end in an optional comma.

```
optTrailingComma
    : /* empty */
    | ","
    ;
```

For example, the following declarations are both legal, and have the same meaning:

```
enum E {
     a, b, c
}

enum E {
     a, b, c,
}
```

This is particularly useful in combination with preprocessor directives:

```
enum E {
#if SUPPORT_A
    a,
#endif
```

```
    b,
    c,
}
```

6.5. Naming conventions

P4 provides a rich assortment of types. Base types include bit-strings, numbers, and errors. There are also built-in types for representing constructs such as parsers, pipelines, actions, and tables. Users can construct new types based on these: structures, enumerations, headers, header stacks, header unions, etc.

In this document we adopt the following conventions:

- Built-in types are written with lowercase characters—e.g., `int<20>`,
- User-defined types are capitalized—e.g., `IPv4Address`,
- Type variables are always uppercase—e.g., `parser P<H, IH>()`,
- Variables are uncapitalized— e.g., `ipv4header`,
- Constants are written with uppercase characters—e.g., `CPU_PORT`, and
- Errors and enumerations are written in camel-case— e.g. `PacketTooShort`.

6.6. P4 programs

A P4 program is a list of declarations:

```
p4program
    : /* empty */
    | p4program declaration
    | p4program ";"  /* empty declaration */
    ;

declaration
    : constantDeclaration
    | externDeclaration
    | actionDeclaration
    | parserDeclaration
    | typeDeclaration
    | controlDeclaration
    | instantiation
    | errorDeclaration
    | matchKindDeclaration
    | functionDeclaration
    ;
```

An empty declarations is indicated with a single semicolon. (Allowing empty declarations accommodates the habits of C/C++ and Java programmers—e.g., certain constructs, like `struct`, do not require a terminating semicolon).

6.6.1. Scopes

Some P4 constructs act as namespaces that create local scopes for names including:

- Derived type declarations (**struct**, **header**, **header_union**, **enum**), which introduce local scopes for field names,
- Block statements, which introduce local lexically-enclosed scopes,
- **parser**, **table**, **action**, and **control** blocks, which introduce local scopes
- Declarations with type variables, which introduce a new scope for those variables. For example, in the following **extern** declaration, the scope of the type variable H extends to the end of the declaration:

```
extern E<H>(/* parameters omitted */) { /* body omitted */ } // scope of H ends here.
```

The order of declarations is important; with the exception of parser states, all uses of a symbol must follow the symbol's declaration. (This is a departure from $P4_{14}$, which allows declarations in any order. This requirement significantly simplifies the implementation of compilers for P4, allowing compilers to use additional information about declared identifiers to resolve ambiguities.)

6.6.2. Stateful elements

Most P4 constructs are stateless: given some inputs they produce a result that solely depends on these inputs. There are only two stateful constructs that may retain information across packets:

- **table**s: Tables are read-only for the data plane, but their entries can be modified by the control-plane,
- **extern** objects: many objects have state that can be read and written by the control plane and data plane. All constructs from the $P4_{14}$ language version that encapsulate state (e.g., counters, meters, registers) are represented using **extern** objects in $P4_{16}$.

In P4 all stateful elements must be explicitly allocated at compilation-time through the process called "instantiation".

In addition, **parser**s, **control** blocks, and **package**s may contain stateful element instantiations. Thus, they are also treated as stateful elements, even if they appear to contain no state, and must be instantiated before they can be used. However, although they are stateful, **table**s do not need to be instantiated explicitly—declaring a **table** also creates an instance of it. This convention is designed to support the common case, since most tables are used just once. To have finer-grained control over when a **table** is instantiated, a programmer can declare it within a **control**.

Recall the example in Section 5.3: TopParser, TopPipe, TopDeparser, Checksum16, and Switch are types. There are two instances of Checksum16, one in TopParser and one in TopDeparser, both called ck. The TopParser, TopDeparser, TopPipe, and Switch are instantiated at the end of the program, in the declaration of the main instance object, which is an instance of the Switch type (a **package**).

6.7. L-values

L-values are expressions that may appear on the left side of an assignment operation or as arguments corresponding to **out** and **inout** function parameters. An l-value represents a storage reference. The following expressions are legal l-values:

```
prefixedNonTypeName
    : nonTypeName
    | dotPrefix nonTypeName
    ;

lvalue
    : prefixedNonTypeName
    | THIS
    | lvalue "." member
    | lvalue "[" expression "]"
    | lvalue "[" expression ":" expression "]"
    | "(" lvalue ")"
    ;
```

- Identifiers of a base or derived type.
- Structure, header, and header union field member access operations (using the dot notation).
- References to elements within header stacks (see Section 8.18): indexing, and references to `last` and `next`.
- The result of a bit-slice operator `[m:l]`.

The following is a legal l-value: `headers.stack[4].field`. Note that method and function calls cannot return l-values.

6.8. Calling convention: call by copy in/copy out

P4 provides multiple constructs for writing modular programs: extern methods, parsers, controls, actions. All these constructs behave similarly to procedures in standard general-purpose programming languages:

- They have named and typed parameters.
- They introduce a new local scope for parameters and local variables.
- They allow arguments to be passed by binding them to their parameters.

Invocations are executed using copy-in/copy-out semantics.

Each parameter may be labeled with a direction:

- **`in`** parameters are read-only. It is an error to use an **`in`** parameter on the left-hand side of an assignment or to pass it to a callee as a non-**`in`** argument. **`in`** parameters are initialized by copying the value of the corresponding argument when the invocation is executed.
- **`out`** parameters are, with a few exceptions listed below, uninitialized and are treated as l-values (See Section 6.7) within the body of the method or function. An argument passed as an **`out`** parameter must be an l-value; after the execution of the call, the value of the parameter is copied to the corresponding storage location for that l-value.
- **`inout`** parameters behave like a combination of **`in`** and **`out`** parameters simultaneously: On entry the value of the arguments is copied to the parameters. On return the value of the parameters is copied back to the arguments. In consequence, an argument passed as an **`inout`** parameter must be an l-value.

- The meaning of parameters with no direction depends upon the kind of entity the parameter is for:
  - For anything other than an action, e.g. a control, parser, or function, a directionless parameter means that the value supplied as an argument in a call must be a compile-time known value (see Section 18.1).
  - For an action, a directionless parameter indicates that it is "action data". See Section 14.1 for the meaning of action data, but its meaning includes the following possibilities:
    * The parameter's value is provided in the P4 program. In this case, the parameter behaves as if the direction were **in**. Such an argument expression need not be a compile-time known value.
    * The parameter's value is provided by the control plane software when an entry is added to a table that uses that action. See Section 14.1.

A directionless parameter of extern object type is passed by reference.

Direction **out** parameters are always initialized at the beginning of execution of the portion of the program that has the **out** parameters, e.g. **control**, **parser**, **action**, function, etc. This initialization is not performed for parameters with any direction that is not **out**.

- If a direction **out** parameter is of type **header** or **header_union**, it is set to "invalid".
- If a direction **out** parameter is of type header stack, all elements of the header stack are set to "invalid", and its `nextIndex` field is initialized to 0 (see Section 8.18).
- If a direction **out** parameter is a compound type, e.g. a struct or tuple, other than one of the types listed above, then apply these rules recursively to its members.
- If a direction **out** parameter has any other type, e.g. **bit**<W>, an implementation need not initialize it to any predictable value.

For example, if a direction **out** parameter has type `s2_t` named `p`:

```
header h1_t {
    bit<8> f1;
    bit<8> f2;
}
struct s1_t {
    h1_t h1a;
    bit<3> a;
    bit<7> b;
}
struct s2_t {
    h1_t h1b;
    s1_t s1;
    bit<5> c;
}
```

then at the beginning of execution of the part of the program that has the **out** parameter `p`, it must be initialized so that `p.h1b` and and `p.s1.h1a` are invalid. No other parts of `p` are required to be initialized.

Arguments are evaluated from left to right prior to the invocation of the function itself. The order of evaluation is important when the expression supplied for an argument can have side-effects. Consider

the following example:

```
extern void f(inout bit x, in bit y);
extern bit g(inout bit z);
bit a;
f(a, g(a));
```

Note that the evaluation of `g` may mutate its argument `a`, so the compiler has to ensure that the value passed to `f` for its first parameter is not changed by the evaluation of the second argument. The semantics for evaluating a function call is given by the following algorithm (implementations can be different as long as they provide the same result):

1. Arguments are evaluated from left to right as they appear in the function call expression.
2. If a parameter has a default value and no corresponding argument is supplied, the default value is used as an argument.
3. For each **`out`** and **`inout`** argument the corresponding l-value is saved (so it cannot be changed by the evaluation of the following arguments). This is important if the argument contains indexing operations into a header stack.
4. The value of each argument is saved into a temporary.
5. The function is invoked with the temporaries as arguments. We are guaranteed that the temporaries that are passed as arguments are never aliased to each other, so this "generated" function call can be implemented using call-by-reference if supported by the architecture.
6. On function return, the temporaries that correspond to **`out`** or **`inout`** arguments are copied in order from left to right into the l-values saved in Step 3.

According to this algorithm, the previous function call is equivalent to the following sequence of statements:

```
bit tmp1 = a;     // evaluate a; save result
bit tmp2 = g(a);  // evaluate g(a); save result; modifies a
f(tmp1, tmp2);    // evaluate f; modifies tmp1
a = tmp1;         // copy inout result back into a
```

To see why Step 3 in the above algorithm is important, consider the following example:

```
header H { bit z; }
H[2] s;
f(s[a].z, g(a));
```

The evaluation of this call is equivalent to the following sequence of statements:

```
bit tmp1 = a;          // save the value of a
bit tmp2 = s[tmp1].z;  // evaluate first argument
bit tmp3 = g(a);       // evaluate second argument; modifies a
f(tmp2, tmp3);         // evaluate f; modifies tmp2
s[tmp1].z = tmp2;      // copy inout result back; dest is not s[a].z
```

When used as arguments, **`extern`** objects can only be passed as directionless parameters—e.g., see the

packet argument in the very simple switch example.

6.8.1. Justification

The main reason for using copy-in/copy-out semantics (instead of the more common call-by-reference semantics) is for controlling the side-effects of **extern** functions and methods. **extern** methods and functions are the main mechanism by which a P4 program communicates with its environment. With copy-in/copy-out semantics **extern** functions cannot hold references to P4 program objects; this enables the compiler to limit the side-effects that **extern** functions may have on the P4 program both in space (they can only affect **out** parameters) and in time (side-effects can only occur at function call time).

In general, **extern** functions are arbitrarily powerful: they can store information in global storage, spawn separate threads, "collude" with each other to share information — but they cannot access any variable in a P4 program. With copy-in/copy-out semantics the compiler can still reason about P4 programs that invoke **extern** functions.

There are additional benefits of using copy-in copy-out semantics:

- It enables P4 to be compiled for architectures that do not support references (e.g., where all data is allocated to named registers. Such architectures may require indices into header stacks that appear in a program to be compile-time known values.)
- It simplifies some compiler analyses, since function parameters can never alias to each other within the function body.

```
parameterList
    : /* empty */
    | nonEmptyParameterList
    ;

nonEmptyParameterList
    : parameter
    | nonEmptyParameterList "," parameter
    ;

parameter
    : optAnnotations direction typeRef name
    | optAnnotations direction typeRef name "=" expression
    ;

direction
    : IN
    | OUT
    | INOUT
    | /* empty */
    ;
```

Following is a summary of the constraints imposed by the parameter directions:

- When used as arguments, extern objects can only be passed as directionless parameters.

- All constructor parameters are evaluated at compilation-time, and in consequence they must all be directionless (they cannot be **in**, **out**, or **inout**); this applies to **package**, **control**, **parser**, and **extern** objects. Expressions for these parameters must be supplied at compile-time, and they must evaluate to compile-time known values. See Section 15 for further details.
- For actions all directionless parameters must be at the end of the parameter list. When an action appears in a **table**'s `actions` list, only the parameters with a direction must be bound. See Section 14.1 for further details.
- Actions can also be explicitly invoked using function call syntax, either from a control block or from another action. In this case, values for all action parameters must be supplied explicitly, including values for the directionless parameters. The directionless parameters in this case behave like **in** parameters. See Section 14.1.1 for further details.
- Default expressions are only allowed for 'in' or direction-less parameters, and the expressions supplied as defaults must be compile-time known values.
- If parameters with default values do not appear at the end of the list of parameters, invocations that use the default values must use named arguments, as in the following example:

```
extern void f(in bit a, in bit<3> b = 2, in bit<5> c);

void g()
{
  f(a = 1, b = 2, c = 3);  // ok
  f(a = 1, c = 3);  // ok, equivalent to the previous call, b uses default value
  f(1, 2, 3);       // ok, equivalent to the previous call
  // f(1, 3); // illegal, since the parameter b is not the last in the list
}
```

6.8.2. Optional parameters

A parameter that is annotated with the `@optional` annotation is optional: the user may omit the value for that parameter in an invocation. Optional parameters can only appear for arguments of: packages, parser types, control types, extern functions, extern methods, and extern object constructors. Optional parameters cannot have default values. If a procedure-like construct has both optional parameters and default values then it can only be called using named arguments. It is recommended, but not mandatory, for all optional parameters to be at the end of a parameter list.

The implementation of such objects is not expressed in P4, so the meaning and implementation of optional parameters should be specified by the target architecture. For example, we can imagine a two-stage switch architecture where the second stage is optional. This could be declared as a package with an optional parameter:

```
package pipeline(/* parameters omitted */);
package switch(pipeline first, @optional pipeline second);

pipeline(/* arguments omitted */) ingress;
switch(ingress) main;   // a switch with a single-stage pipeline
```

Here the target architecture could implement the elided optional argument using an empty pipeline.

The following example shows optional parameters and parameters with default values.

```
extern void h(in bit<32> a, in bool b = true);  // default value

// function calls
h(10);  // same as h(10, true);
h(a = 10);  // same as h(10, true);
h(a = 10, b = true);

struct Empty {}
control nothing(inout Empty h, inout Empty m) {
   apply {}
}

parser parserProto<H, M>(packet_in p, out H h, inout M m);
control controlProto<H, M>(inout H h, inout M m);

package pack<HP, MP, HC, MC>(@optional parserProto<HP, MP> _parser,  // optional parameter
                             controlProto<HC, MC> _control = nothing()); // default parameter value

pack() main;   // No value for _parser, _control is an instance of nothing()
```

6.9. Name resolution

P4 objects that introduce namespaces are organized in a hierarchical fashion. There is a top-level unnamed namespace containing all top-level declarations.

Identifiers prefixed with a dot are always resolved in the top-level namespace.

```
const bit<32> x = 2;
control c() {
   int<32> x = 0;
   apply {
       x = x + (int<32>).x;  // x is the int<32> local variable,
                             // .x is the top-level bit<32> variable
   }
}
```

References to resolve an identifier are attempted inside-out, starting with the current scope and proceeding to all lexically enclosing scopes. The compiler may provide a warning if multiple resolutions are possible for the same name (name shadowing).

```
const bit<4> x = 1;
control p() {
    const bit<8> x = 8;    // x declaration shadows global x
    const bit<4> y = .x;   // reference to top-level x
```

```
    const bit<8> z = x;    // reference to p's local x
    apply {}
}
```

6.10. Visibility

Identifiers defined in the top-level namespace are globally visible. Declarations within a **parser** or **control** are private and cannot be referred to from outside of the enclosing **parser** or **control**.

7. P4 data types

$P4_{16}$ is a statically-typed language. Programs that do not pass the type checker are considered invalid and rejected by the compiler. P4 provides a number of base types as well as type operators that construct derived types. Some values can be converted to a different type using casts. However, to make user intents clear, implicit casts are only allowed in a few circumstances and the range of casts available is intentionally restricted.

7.1. Base types

P4 supports the following built-in base types:

- The **void** type, which has no values and can be used only in a few restricted circumstances.
- The **error** type, which is used to convey errors in a target-independent, compiler-managed way.
- The **string** type, which can be used with compile-time known values of type string.
- The **match_kind** type, which is used for describing the implementation of table lookups,
- **bool**, which represents Boolean values
- **int**, which represents arbitrary-sized integer values
- Bit-strings of fixed width, denoted by **bit**<>
- Fixed-width signed integers represented using two's complement **int**<>
- Bit-strings of dynamically-computed width with a fixed maximum width **varbit**<>

```
baseType
    : BOOL
    | MATCH_KIND
    | ERROR
    | BIT
    | STRING
    | INT
    | BIT "<" INTEGER ">"
    | INT "<" INTEGER ">"
    | VARBIT "<" INTEGER ">"
    | BIT "<" "(" expression ")" ">"
    | INT "<" "(" expression ")" ">"
    | VARBIT "<" "(" expression ")" ">"
    ;
```

7.1.1. The void type

The void type is written `void`. It contains no values. It is not included in the production rule `baseType` as it can only appear in few restricted places in P4 programs.

7.1.2. The error type

The error type contains opaque distinct values that can be used to signal errors. It is written as `error`. New elements of the error type are defined with the syntax:

```
errorDeclaration
    : ERROR "{" identifierList "}"
    ;
```

All elements of the `error` type are inserted into the `error` namespace, irrespective of the place where an error is defined. `error` is similar to an enumeration (`enum`) type in other languages. A program can contain multiple `error` declarations, which the compiler will merge together. It is an error to declare the same identifier multiple times. Expressions of type `error` are described in Section 8.2.

For example, the following declaration creates two elements of the `error` type (these errors are declared in the P4 core library):

```
error { ParseError, PacketTooShort }
```

The underlying representation of errors is target-dependent.

7.1.3. The match kind type

The `match_kind` type is very similar to the `error` type and is used to declare a set of distinct names that may be used in a table's key property (described in Section 14.2.1). All identifiers are inserted into the top-level namespace. It is an error to declare the same `match_kind` identifier multiple times.

```
matchKindDeclaration
    : MATCH_KIND "{" identifierList optTrailingComma "}"
    ;
```

The P4 core library contains the following match_kind declaration:

```
match_kind {
   exact,
   ternary,
   lpm
}
```

Architectures may support additional `match_kind`s. The declaration of new `match_kind`s can only occur within model description files; P4 programmers cannot declare new match kinds.

Operations on values of type `match_kind` are described in Section 8.4.

7.1.4. The Boolean type

The Boolean type `bool` contains just two values, `false` and `true`. Boolean values are not integers or bit-strings.

7.1.5. Strings

The type `string` represents strings. There are no operations on string values; one cannot declare variables with a `string` type. Parameters with type `string` can be only directionless (see Section 6.8). P4 does not support string manipulation in the dataplane; the `string` type is only allowed for describing compile-time known values (i.e., string literals, as discussed in Section 6.4.3.3). Even so, the string type is useful, for example, in giving the type signature for extern functions such as the following:

```
extern void log(string message);
```

As another example, the following annotation indicates that the specified name should be used for a given table in the generated control-plane API:

```
@name("acl") table t1 { /* body omitted */ }
```

7.1.6. Integers (signed and unsigned)

P4 supports arbitrary-size integer values. The typing rules for the integer types are chosen according to the following principles:

- Inspired by C: Typing of integers is modeled after the well-defined parts of C, expanded to cope with arbitrary fixed-width integers. In particular, the type of the result of an expression only depends on the expression operands, and not on how the result of the expression is consumed.
- No undefined behaviors: P4 attempts to avoid many of C's behaviors, which include the size of an integer (int), the results produced on overflow, and the results produced for some input combinations (e.g., shifts with negative amounts, overflows on signed numbers, etc.). P4 computations on integer types have no undefined behaviors.
- Least surprise: The P4 typing rules are chosen to behave as closely as possible to traditional well-behaved C programs.
- Forbid rather than surprise: Rather than provide surprising or undefined results (e.g., in C comparisons between signed and unsigned integers), we have chosen to forbid expressions with ambiguous interpretations. For example, P4 does not allow binary operations that combine signed and unsigned integers.

The priority of arithmetic operations is identical to C—e.g., multiplication binds tighter than addition.

7.1.6.1. Portability No P4 target can support all possible types and operations. For example, the type `bit<23132312>` is legal in P4, but it is highly unlikely to be supported on any target in practice. Hence, each target can impose restrictions on the types it can support. Such restrictions may include:

- The maximum width supported
- Alignment and padding constraints (e.g., arithmetic may only be supported on widths which are an integral number of bytes).

- Constraints on some operands (e.g., some architectures may only support multiplications by small values, or shifts with small values).

The documentation supplied with a target should clearly specify restrictions, and target-specific compilers should provide clear error messages when such restrictions are encountered. An architecture may reject a well-typed P4 program and still be conformant to the P4 spec. However, if an architecture accepts a P4 program as valid, the runtime program behavior should match this specification.

7.1.6.2. Unsigned integers (bit-strings) An unsigned integer (which we also call a "bit-string") has an arbitrary width, expressed in bits. A bit-string of width `W` is declared as: `bit<W>`. `W` must be an expression that evaluates to a local compile-time known value (see Section 18.1) that is a non-negative integer. When using an expression for the size, the expression must be parenthesized. Bitstrings with width 0 are allowed; they have no actual bits, and can only have the value 0. See 8.25 for additional details. Note that `bit<W>` type refers to both cases of `bit<W>` and `bit<(expression)>` where the width is a compile-time known value.

```
const bit<32> x = 10;   // 32-bit constant with value 10.
const bit<(x + 2)> y = 15;  // 12-bit constant with value 15.
                            // expression for width must use ()
```

Bits within a bit-string are numbered from `0` to `W-1`. Bit `0` is the least significant, and bit `W-1` is the most significant.

For example, the type `bit<128>` denotes the type of bit-string values with 128 bits numbered from 0 to 127, where bit 127 is the most significant.

The type `bit` is a shorthand for `bit<1>`.

P4 architectures may impose additional constraints on bit types: for example, they may limit the maximum size, or they may only support some arithmetic operations on certain sizes (e.g., 16-, 32-, and 64- bit values).

All operations that can be performed on unsigned integers are described in Section 8.6.

7.1.6.3. Signed Integers Signed integers are represented using two's complement. An integer with `W` bits is declared as: `int<W>`. `W` must be an expression that evaluates to a local compile-time known (see Section 18.1) value that is a non-negative integer. Note that `int<W>` type refers to both cases of `int<W>` and `int<(expression)>` where the width is a local compile-time known value.

Bits within an integer are numbered from `0` to `W-1`. Bit `0` is the least significant, and bit `W-1` is the sign bit.

For example, the type `int<64>` describes the type of integers represented using exactly 64 bits with bits numbered from 0 to 63, where bit 63 is the most significant (sign) bit.

P4 architectures may impose additional constraints on signed types: for example, they may limit the maximum size, or they may only support some arithmetic operations on certain sizes (e.g., 16-, 32-, and 64- bit values).

All operations that can be performed on signed integers are described in Section 8.7.

A signed integer with width 1 can only have two legal values: 0 and -1.

7.1.6.4. Dynamically-sized bit-strings Some network protocols use fields whose size is only known at runtime (e.g., IPv4 options). To support restricted manipulations of such values, P4 provides

a special bit-string type whose size is set at runtime, called a **varbit**.

The type **varbit**<W> denotes a bit-string with a width of at most W bits, where W is a local compile-time known value (see Section 18.1) that is a non-negative integer. For example, the type **varbit**<120> denotes the type of bit-string values that may have between 0 and 120 bits. Most operations that are applicable to fixed-size bit-strings (unsigned numbers) cannot be performed on dynamically sized bit-strings. Note that **varbit**<W> type refers to both cases of **varbit**<W> and **varbit**<(expression)> where the width is a compile-time known value.

P4 architectures may impose additional constraints on varbit types: for example, they may limit the maximum size, or they may require **varbit** values to always contain an integer number of bytes at runtime.

All operations that can be performed on varbits are described in Section 8.10.

7.1.6.5. Arbitrary-precision integers

The arbitrary-precision data type describes integers with an unlimited precision. This type is written as **int**.

This type is reserved for integer literals and expressions that involve only literals. No P4 runtime value can have an **int** type; at compile time the compiler will convert all int values that have a runtime component to fixed-width types, according to the rules described below.

All operations that can be performed on arbitrary-precision integers are described in Section 8.8. The following example shows three constant definitions whose values are arbitrary-precision integers.

```
const int a = 5;
const int b = 2 * a;
const int c = b - a + 3;
```

Parameters with type **int** are not supported for actions. Parameters with type **int** for other callable entities of a program, e.g. controls, parsers, or functions, must be directionless, indicating that all calls must provide a compile-time known value as an argument for such a parameter. See Section 6.8 for more details on directionless parameters.

7.1.6.6. Integer literal types

The types of integer literals are as follows:

- An integer with no type prefix has type **int**.
- A non-negative integer prefixed with an integer width W and the character w has type **bit**<W>.
- An integer prefixed with an integer width W and the character s has type **int**<W>.

The table below shows several examples of integer literals and their types. For additional examples of literals see Section 6.4.3.

Literal	Interpretation
10	Type is **int**, value is 10
8w10	Type is **bit**<8>, value is 10
8s10	Type is **int**<8>, value is 10
2s3	Type is **int**<2>, value is -1 (last 2 bits), overflow warning
1w10	Type is **bit**<1>, value is 0 (last bit), overflow warning
1s1	Type is **int**<1>, value is -1, overflow warning

7.2. Derived types

P4 provides a number of type constructors that can be used to derive additional types including:

- **enum**
- **header**
- header stacks
- **struct**
- **header_union**
- **tuple**
- type specialization
- **extern**
- **parser**
- **control**
- **package**

The types **header**, **header_union**, **enum**, **struct**, **extern**, **parser**, **control**, and **package** can only be used in type declarations, where they introduce a new name for the type. The type can subsequently be referred to using this identifier.

Other types cannot be declared, but are synthesized by the compiler internally to represent the type of certain language constructs. These types are described in Section 7.2.9: set types and function types. For example, the programmer cannot declare a variable with type "set", but she can write an expression whose value evaluates to a `set` type. These types are used during type-checking.

```
typeDeclaration
    : derivedTypeDeclaration
    | typedefDeclaration ";"
    | parserTypeDeclaration ";"
    | controlTypeDeclaration ";"
    | packageTypeDeclaration ";"
    ;

derivedTypeDeclaration
    : headerTypeDeclaration
    | headerUnionDeclaration
    | structTypeDeclaration
    | enumDeclaration
    ;

typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    | p4listType
    | tupleType
    ;
```

```
namedType
    : typeName
    | specializedType
    ;

prefixedType
    : TYPE_IDENTIFIER
    | dotPrefix TYPE_IDENTIFIER
    ;

typeName
    : prefixedType
    ;
```

7.2.1. Enumeration types

An enumeration type is defined using the following syntax:

```
enumDeclaration
    : optAnnotations ENUM name "{" identifierList optTrailingComma "}"
    | optAnnotations ENUM typeRef name "{"
      specifiedIdentifierList optTrailingComma "}"
    ;

identifierList
    : name
    | identifierList "," name
    ;

specifiedIdentifierList
    : specifiedIdentifier
    | specifiedIdentifierList "," specifiedIdentifier
    ;

specifiedIdentifier
    : name "=" initializer
    ;
```

For example, the declaration

```
enum Suits { Clubs, Diamonds, Hearths, Spades }
```

introduces a new enumeration type, which contains four elements—e.g., `Suits.Clubs`. An **`enum`** declaration introduces a new identifier in the current scope for naming the created type along with its distinct elements. The underlying representation of the `Suits` enum is not specified, so their "size" in bits is not

specified (it is target-specific).

It is also possible to specify an **enum** with an underlying representation. These are sometimes called serializable **enum**s, because headers are allowed to have fields with such **enum** types. This requires the programmer provide both the fixed-width unsigned (or signed) integer type and an associated integer value for each symbolic entry in the enumeration. The symbol `typeRef` in the grammar above must be one of the following types:

- an unsigned integer, i.e. **bit**<W> for some local compile-time known value `W`.
- a signed integer, i.e. **int**<W> for some local compile-time known value `W`.
- a type name declared via **typedef**, where the base type of that type is either one of the types listed above, or another **typedef** name that meets these conditions. For example, the declaration

```
enum bit<16> EtherType {
  VLAN      = 0x8100,
  QINQ      = 0x9100,
  MPLS      = 0x8847,
  IPV4      = 0x0800,
  IPV6      = 0x86dd
}
```

introduces a new enumeration type, which contains five elements—e.g., `EtherType.IPV4`. This **enum** declaration specifies the fixed-width unsigned integer representation for each entry in the **enum** and provides an underlying type: **bit**<16>. This kind of **enum** declaration can be thought of as declaring a new **bit**<16> type, where variables or fields of this type are expected to be unsigned 16-bit integer values, and the mapping of symbolic to numeric values defined by the **enum** are also defined as a part of this declaration. In this way, an **enum** with an underlying type can be thought of as being a type derived from the underlying type carrying equality, assignment, and casts to/from the underlying type.

Compiler implementations are expected to raise an error if the fixed-width integer representation for an enumeration entry falls outside the representation range of the underlying type.

For example, the declaration

```
enum bit<8> FailingExample {
  first           = 1,
  second          = 2,
  third           = 3,
  unrepresentable = 300
}
```

would raise an error because 300, the value associated with `FailingExample.unrepresentable` cannot be represented as a **bit**<8> value.

The `initializer` expression must be a compile-time known value.

Annotations, represented by the non-terminal `optAnnotations`, are described in Section 20.

Operations on **enum** values are described in Section 8.3.

7.2.2. Header types

The declaration of a **header** type is given by the following syntax:

```
headerTypeDeclaration
    : optAnnotations HEADER name optTypeParameters "{" structFieldList "}"
    ;

structFieldList
    : /* empty */
    | structFieldList structField
    ;

structField
    : optAnnotations typeRef name ";"
    ;
```

where each `typeRef` is restricted to a bit-string type (fixed or variable), a fixed-width signed integer type, a boolean type, or a struct that itself contains other struct fields, nested arbitrarily, as long as all of the "leaf" types are **`bit`**`<W>`, **`int`**`<W>`, a serializable **`enum`**, or a **`bool`**. If a **`bool`** is used inside a P4 header, all implementations encode the **`bool`** as a one bit long field, with the value `1` representing **`true`** and `0` representing **`false`**. Field names have to be distinct.

A header declaration introduces a new identifier in the current scope; the type can be referred to using this identifier. A header is similar to a **`struct`** in C, containing all the specified fields. However, in addition, a header also contains a hidden Boolean "validity" field. When the "validity" bit is **`true`** we say that the "header is valid". When a local variable with a header type is declared, its "validity" bit is automatically set to **`false`**. The "validity" bit can be manipulated by using the header methods `isValid()`, `setValid()`, and `setInvalid()`, as described in Section 8.17.

Note, nesting of headers is not supported. One reason is that it leads to complications in defining the behavior of arbitrary sequences of `setValid`, `setInvalid`, and `emit` operations. Consider an example where header `h1` contains header `h2` as a member, both currently valid. A program executes `h2.setInvalid()` followed by `packet.emit(h1)`. Should all fields of `h1` be emitted, but skipping `h2`? Similarly, should `h1.setInvalid()` invalidate all headers contained within `h1`, regardless of how deeply they are nested?

Header types may be empty:

```
header Empty_h { }
```

Note that an empty header still contains a validity bit.

When a struct is inside of a header, the order of the fields for the purposes of `extract` and `emit` calls is the order of the fields as defined in the source code. An example of a header including a struct is included below.

```
struct ipv6_addr {
    bit<32> Addr0;
    bit<32> Addr1;
    bit<32> Addr2;
    bit<32> Addr3;
}
```

```
header ipv6_t {
    bit<4>    version;
    bit<8>    trafficClass;
    bit<20>   flowLabel;
    bit<16>   payloadLen;
    bit<8>    nextHdr;
    bit<8>    hopLimit;
    ipv6_addr src;
    ipv6_addr dst;
}
```

Headers that do not contain any **varbit** field are "fixed size." Headers containing **varbit** fields have "variable size." The size (in bits) of a fixed-size header is simply the sum of the sizes of all component fields (without counting the validity bit). There is no padding or alignment of the header fields. Targets may impose additional constraints on header types—e.g., restricting headers to sizes that are an integer number of bytes.

For example, the following declaration describes a typical Ethernet header:

```
header Ethernet_h {
   bit<48> dstAddr;
   bit<48> srcAddr;
   bit<16> etherType;
}
```

The following variable declaration uses the newly introduced type `Ethernet_h`:

```
Ethernet_h ethernetHeader;
```

P4's parser language provides an `extract` method that can be used to "fill in" the fields of a header from a network packet, as described in Section 13.8. The successful execution of an `extract` operation also sets the validity bit of the extracted header to **true**.

Here is an example of an IPv4 header with variable-sized options:

```
header IPv4_h {
   bit<4>       version;
   bit<4>       ihl;
   bit<8>       diffserv;
   bit<16>      totalLen;
   bit<16>      identification;
   bit<3>       flags;
   bit<13>      fragOffset;
   bit<8>       ttl;
   bit<8>       protocol;
   bit<16>      hdrChecksum;
   bit<32>      srcAddr;
   bit<32>      dstAddr;
   varbit<320>  options;
```

```
}
```

As demonstrated by a code example in Section 13.8.2, another way to support headers that contain variable-length fields is to define two headers – one fixed length, one containing a **varbit** field – and extract each part in separate parsing steps.

Notice that the names `isValid`, `setValid`, `minSizeInBits`, etc. are all valid header field names.

7.2.3. Header stacks

A header stack represents an array of headers or header unions. A header stack type is defined as:

```
headerStackType
    : typeName "[" expression "]"
    | specializedType "[" expression "]"
    ;
```

where `typeName` is the name of a header or header union type. For a header stack `hs[n]`, the term `n` is the maximum defined size, and must be a local compile-time known value that is a positive integer. Nested header stacks are not supported. At runtime a stack contains `n` values with type `typeName`, only some of which may be valid. Expressions on header stacks are discussed in Section 8.18.

For example, the following declarations,

```
header Mpls_h {
    bit<20> label;
    bit<3>  tc;
    bit     bos;
    bit<8>  ttl;
}
Mpls_h[10] mpls;
```

introduce a header stack called `mpls` containing ten entries, each of type `Mpls_h`.

Operations on header stacks are described in Section 8.18.

7.2.4. Header unions

A header union represents an alternative containing at most one of several different headers. Header unions can be used to represent "options" in protocols like TCP and IP. They also provide hints to P4 compilers that only one alternative will be present, allowing them to conserve storage resources.

A header union is defined as:

```
headerUnionDeclaration
    : optAnnotations HEADER_UNION name optTypeParameters "{" structFieldList "}"
    ;
```

This declaration introduces a new type with the specified name in the current scope. Each element of the list of fields used to declare a header union must be of header type. An empty list of fields is legal. Field names have to be distinct.

As an example, the type `Ip_h` below represents the union of an IPv4 and IPv6 headers:

```
header_union IP_h {
  IPv4_h v4;
  IPv6_h v6;
}
```

A header union is not considered a type with fixed length.

Operation on header unions are described in Section 8.19.

7.2.5. Struct types

P4 `struct` types are defined with the following syntax:

```
structTypeDeclaration
    : optAnnotations STRUCT name optTypeParameters "{" structFieldList "}"
    ;
```

This declaration introduces a new type with the specified name in the current scope. Field names have to be distinct. An empty struct (with no fields) is legal. For example, the structure `Parsed_headers` below contains the headers recognized by a simple parser:

```
header Tcp_h { /* fields omitted */ }
header Udp_h { /* fields omitted */ }
struct Parsed_headers {
    Ethernet_h ethernet;
    Ip_h       ip;
    Tcp_h      tcp;
    Udp_h      udp;
}
```

7.2.6. Tuple types

A tuple is similar to a `struct`, in that it holds multiple values. The type of tuples with n component types `T1`,...,`Tn` is written as

```
tuple<T1,/* more fields omitted */,Tn>
```

```
tupleType
    : TUPLE "<" typeArgumentList ">"
    ;
```

Operations that manipulate tuple types are described in Section 8.12.

The type `tuple<>` is a tuple type with no components.

7.2.7. List types

A list holds zero or more values, where every element must have the same type. The type of a list where all elements have type `T` is written as

```
list<T>
```

```
p4listType
    : LIST "<" typeArg ">"
    ;
```

Operations that manipulate list types are described in Section 8.14.

7.2.8. Type nesting rules

The table below lists all types that may appear as members of headers, header unions, structs, tuples, and lists. Note that **int** by itself (i.e. not as part of an **int**<N> type expression) means an arbitrary-precision integer, without a width specified.

	Container type				
Element type	header	header_union	struct or tuple	list	header stack
bit<W>	allowed	error	allowed	allowed	error
int<W>	allowed	error	allowed	allowed	error
varbit<W>	allowed	error	allowed	allowed	error
int	error	error	error	allowed	error
void	error	error	error	error	error
string	error	error	error	allowed	error
error	error	error	allowed	allowed	error
match_kind	error	error	error	allowed	error
bool	allowed	error	allowed	allowed	error
enum	allowed[1]	error	allowed	allowed	error
header	error	allowed	allowed	allowed	allowed
header stack	error	error	allowed	allowed	error
header_union	error	error	allowed	allowed	allowed
struct	allowed[2]	error	allowed	allowed	error
tuple	error	error	allowed	allowed	error
list	error	error	error	allowed	error

Rationale: **int** does not have precise storage requirements, unlike **bit**<> or **int**<> types. **match_kind** values are not useful to store in a variable, as they are only used to specify how to match fields in table search keys, which are all declared at compile time. **void** is not useful as part of another data structure. Headers must have precisely defined formats as sequences of bits in order for them to be parsed or deparsed.

[1] an **enum** type used as a field in a **header** must specify a underlying type and representation for **enum** elements.

[2] a **struct** or nested **struct** type that has the same properties, used as a field in a **header** must contain only **bit**<W>, **int**<W>, a serializable **enum**, or a **bool**.

Note the two-argument `extract` method (see Section 13.8.2) on packets only supports a single **`varbit`** field in a header.

The table below lists all types that may appear as base types in a **`typedef`** or **`type`** declaration.

Base type B	**`typedef B <name>`**	**`type B <name>`**
`bit`<W>	allowed	allowed
`int`<W>	allowed	allowed
`varbit`<W>	allowed	error
`int`	allowed	error
`void`	error	error
`error`	allowed	error
`match_kind`	error	error
`bool`	allowed	allowed
`enum`	allowed	error
`header`	allowed	error
header stack	allowed	error
`header_union`	allowed	error
`struct`	allowed	error
`tuple`	allowed	error
a **`typedef`** name	allowed	allowed[3]
a **`type`** name	allowed	allowed

7.2.9. Synthesized data types

For the purposes of type-checking the P4 compiler can synthesize some type representations which cannot be directly expressed by users. These are described in this section: set types and function types.

7.2.9.1. Set types The type `set<T>` describes sets of values of some type `T`. Set types can only appear in restricted contexts in P4 programs. For example, the range expression `8w5 .. 8w8` describes a set containing the 8-bit numbers 5, 6, 7, and 8, so its type is `set<bit<8>>;`. This expression can be used as a label in a **`select`** expression (see Section 13.6), matching any value in this range. Set types cannot be named or declared by P4 programmers, they are only synthesized by the compiler internally and used for type-checking. Expressions with set types are described in Section 8.15.

7.2.9.2. Function types Function types are created by the P4 compiler internally to represent the types of functions (explicit functions or extern functions) and methods during type-checking. We also call the type of a function its signature. Libraries can contain functions and extern function declarations.

For example, consider the following declarations:

```
extern void random(in bit<5> logRange, out bit<32> value);

bit<32> add(in bit<32> left, in bit<32> right) {
```

[3] **`type`** `B` `<name>` is allowed for a type name `B` defined via **`typedef`** `X B` if **`type`** `X` `<name>` is allowed.

```
    return left + right;
}
```

These declarations describe two objects:

- `random`, which has a function type, representing the following information:
  - the result type is **`void`**
  - the function has two inputs
  - the first formal parameter has direction **`in`**, type **`bit`**<5>, and name `logRange`
  - the second formal parameter has direction **`out`**, type **`bit`**<32>, and name `value`
- `add`, also has a function type, representing the following information:
  - the result type is **`bit`**<32>
  - the function has two inputs
  - both inputs have direction **`in`** and type **`bit`**<32>

7.2.10. Extern types

P4 supports extern object declarations and extern function declarations using the following syntax.

```
externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters "{" methodPrototypes "}"
    | optAnnotations EXTERN functionPrototype ";"
    ;
```

7.2.10.1. Extern functions An extern function declaration describes the name and type signature of the function, but not its implementation.

```
functionPrototype
    : typeOrVoid name optTypeParameters "(" parameterList ")"
    ;
```

For an example of an **`extern`** function declaration, see Section 7.2.9.2.

7.2.10.2. Extern objects An extern object declaration declares an object and all methods that can be invoked to perform computations and to alter the state of the object. Extern object declarations can also optionally declare constructor methods; these must have the same name as the enclosing **`extern`** type, no type parameters, and no return type. Extern declarations may only appear as allowed by the architecture model and may be specific to a target.

```
methodPrototypes
    : /* empty */
    | methodPrototypes methodPrototype
    ;
```

```
methodPrototype
    : optAnnotations functionPrototype ';'
    | optAnnotations TYPE_IDENTIFIER '(' parameterList ')' ';' //constructor
    | optAnnotations ABSTRACT functionPrototype ";"
    ;

typeOrVoid
    : typeRef
    | VOID
    | IDENTIFIER     // may be a type variable
    ;

optTypeParameters
    : /* empty */
    | typeParameters
    ;

typeParameters
    : "<" typeParameterList ">"
    ;

typeParameterList
    : name
    | typeParameterList "," name
    ;
```

For example, the P4 core library introduces two extern objects `packet_in` and `packet_out` used for manipulating packets (see Sections 13.8 and 16). Here is an example showing how the methods of these objects can be invoked on a packet:

```
extern packet_out {
    void emit<T>(in T hdr);
}
control d(packet_out b, in Hdr h) {
    apply {
        b.emit(h.ipv4);       // write ipv4 header into output packet
    }                         // by calling emit method
}
```

Functions and methods are the only P4 constructs that support overloading: there can exist multiple methods with the same name in the same scope. When overloading is used, the compiler must be able to disambiguate at compile-time which method or function is being called, either by the number of arguments or by the names of the arguments, when calls are specifying argument names. Argument type information is not used in disambiguating calls.

Notice that overloading of parsers, controls, or packages is not allowed:

```
parser p(packet_in p, out bit<32> value) {
  ...
}

// The following will cause an error about a duplicate declaration
//parser p(packet_in p, out Headers headers) {
//   ...
//}
```

Abstract methods Typical extern object methods are built-in, and are implemented by the target architecture. P4 programmers can only call such methods.

However, some types of extern objects may provide methods that can be implemented by the P4 programmers. Such methods are described with the **abstract** keyword prior to the method definition. Here is an example:

```
extern Balancer {
    Balancer();
    // get the number of active flows
    bit<32> getFlowCount();
    // return port index used for load-balancing
    // @param address: IPv4 source address of flow
    abstract bit<4> on_new_flow(in bit<32> address);
}
```

When such an object is instantiated the user has to supply an implementation of all the **abstract** methods (see 11.3.1).

7.2.11. Type specialization

A generic type may be specialized by specifying arguments for its type variables. In cases where the compiler can infer type arguments type specialization is not necessary. When a type is specialized all its type variables must be bound.

```
specializedType
    : typeName "<" typeArgumentList ">"
    ;
```

For example, the following extern declaration describes a generic block of registers, where the type of the elements stored in each register is an arbitrary `T`.

```
extern Register<T> {
    Register(bit<32> size);
    T read(bit<32> index);
    void write(bit<32> index, T value);
}
```

The type `T` has to be specified when instantiating a set of registers, by specializing the Register type:

```
Register<bit<32>>(128) registerBank;
```

The instantiation of `registerBank` is made using the `Register` type specialized with the `bit<32>` bound to the `T` type argument.

`struct`, `header`, `header_union` and header stack types can be generic as well. In order to use such a generic type it must be specialized with appropriate type arguments. For example

```
// generic structure type
struct S<T> {
    T field;
    bool valid;
}

struct G<T> {
    S<T> s;
}

// specialize S by replacing 'T' with 'bit<32>'
const S<bit<32>> s = { field = 32w0, valid = false };
// Specialize G by replacing 'T' with 'bit<32>'
const G<bit<32>> g = { s = { field = 0, valid = false } };

// generic header type
header H<T> {
    T field;
}

// Specialize H by replacing 'T' with 'bit<8>'
const H<bit<8>> h = { field = 1 };
// Header stack produced from a specialization of a generic header type
H<bit<8>>[10] stack;

// Generic header union
header_union HU<T> {
    H<bit<32>> h32;
    H<bit<8>>  h8;
    H<T>       ht;
}

// Header union with a type obtained by specializing a generic header union type
HU<bit> hu;
```

7.2.12. Parser and control blocks types

Parsers and control blocks types are similar to function types: they describe the signature of parsers and control blocks. Such functions have no return values. Declarations of parsers and control block types in architectures may be generic (i.e., have type parameters).

The types **parser**, **control**, and **package** cannot be used as types of arguments for methods, parsers, controls, tables, or actions. They can be used as types for the arguments passed to constructors (see Section 15).

7.2.12.1. Parser type declarations

A parser type declaration describes the signature of a parser. A parser should have at least one argument of type `packet_in`, representing the received packet that is processed.

```
parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters
      "(" parameterList ")"
    ;
```

For example, the following is a type declaration of a parser type named `P` that is parameterized on a type variable `H`. That parser receives as input a `packet_in` value `b` and produces two values:

- A value with a user-defined type `H`
- A value with a predefined type `Counters`

```
struct Counters { /* Fields omitted */ }
parser P<H>(packet_in b,
            out H packetHeaders,
            out Counters counters);
```

7.2.12.2. Control type declarations

A control type declaration describes the signature of a control block.

```
controlTypeDeclaration
    : optAnnotations CONTROL name optTypeParameters
      "(" parameterList ")"
    ;
```

Control type declarations are similar to parser type declarations.

7.2.13. Package types

A package type describes the signature of a package.

```
packageTypeDeclaration
    : optAnnotations PACKAGE name optTypeParameters
      "(" parameterList ")"
```

```
    ;
```

All parameters of a package are evaluated at compilation time, and in consequence they must all be directionless (they cannot be **in**, **out**, or **inout**). Otherwise package types are very similar to parser type declarations. Packages can only be instantiated; there are no runtime behaviors associated with them.

7.2.14. Don't care types

A don't care (underscore, "_") can be used in some circumstances as a type. It should be only used in a position where one could write a bound type variable. The underscore can be used to reduce code complexity—when it is not important what the type variable binds to (during type unification the don't care type can unify with any other type). An example is given Section 17.1.

7.3. Default values

Some P4 types define a "default value," which can be used to automatically initialize values of that type. The default values are as follows:

- For **int**, **bit**<N> and **int**<N> types the default value is 0.
- For **bool** the default value is **false**.
- For **error** the default value is **error**.NoError (defined in core.p4)
- For **string** the default value is the empty string ""
- For **varbit**<N> the default value is a string of zero bits (there is currently no P4 literal to represent such a value).
- For **enum** values with an underlying type the default value is 0, even if 0 is actually not one of the named values in the enum.
- For **enum** values without an underlying type the default value is the first value that appears in the **enum** type declaration.
- For **header** types the default value is invalid.
- For header stacks the default value is that all elements are invalid and the nextIndex is 0.
- For **header_union** values the default value is that all union elements are invalid.
- For **struct** types the default value is a **struct** where each field has the default value of the suitable field type – if all such default values are defined.
- For a **tuple** type the default value is a **tuple** where each field has the default value of the suitable type – if all such default values are defined.

Note that some types do not have default values, e.g., **match_kind**, set types, function types, extern types, parser types, control types, package types.

7.4. Numeric types

Many P4 operations are restrained to expressions that evaluate to numeric values. Such expressions must have one of the following numeric types:

- **int** - an arbitrary-precision integer (section 7.1.6.5)
- **bit**<W> - a W-bit unsigned integer where W >= 0 (section 7.1.6.2)
- **int**<W> - a W-bit signed integer where W >= 1 (section 7.1.6.3)
- a serializable **enum** with an underlying type that is **bit**<W> or **int**<W> (section 7.2.1).

7.5. typedef

A **typedef** declaration can be used to give an alternative name to a type.

```
typedefDeclaration
    : optAnnotations TYPEDEF typeRef name ';'
    | optAnnotations TYPEDEF derivedTypeDeclaration name ';'
    ;
```

```
typedef bit<32> u32;
typedef struct Point { int<32> x; int<32> y; } Pt;
typedef Empty_h[32] HeaderStack;
```

The two types are treated as synonyms, and all operations that can be executed using the original type can be also executed using the newly created type.

If **typedef** is used with a generic type the type must be specialized with the suitable number of type arguments:

```
struct S<T> {
   T field;
}

// typedef S X;  -- illegal: S does not have type arguments
typedef S<bit<32>> X;  // -- legal
```

7.6. Introducing new types

Similarly to **typedef**, the keyword **type** can be used to introduce a new type.

```
    | optAnnotations TYPE typeRef name
```

```
type bit<32> U32;
U32 x = (U32)0;
```

While similar to **typedef**, the **type** keyword introduces a new type which is not a synonym with the original type: values of the original type and the newly introduced type cannot be mixed in expressions.

Currently the types that can be created by the **type** keyword are restricted to one of: **bit**<>, **int**<>, **bool**, or types defined using **type** from such types.

One important use of such types is in describing P4 values that need to be exchanged with the control plane through communication channels (e.g., through the control-plane API or through network packets sent to the control plane). For example, a P4 architecture may define a type for the switch ports:

```
type bit<9> PortId_t;
```

This declaration will prevent `PortId_t` values from being used in arithmetic expressions without casts.

Moreover, this declaration may enable special manipulation or such values by software that lies outside of the datapath (e.g., a target-specific toolchain could include software that automatically converts values of type `PortId_t` to a different representation when exchanged with the control-plane software).

8. Expressions

This section describes all expressions that can be used in P4, grouped by the type of value they produce.

The grammar production rule for general expressions is as follows:

```
expression
    : INTEGER
    | DOTS                          // DOTS is ...
    | STRING_LITERAL
    | TRUE
    | FALSE
    | prefixedNonTypeName
    | expression '[' expression ']'
    | expression '[' expression ':' expression ']'
    | '{' expressionList optTrailingComma '}'
    | "{#}"
    | '{' kvList optTrailingComma '}'
    | "{" kvList "," DOTS optTrailingComma "}"
    | '(' expression ')'
    | '!' expression
    | '~' expression
    | '-' expression
    | '+' expression
    | typeName '.' member
    | ERROR '.' member
    | expression '.' member
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '+' expression
    | expression '-' expression
    | expression '|+|' expression
    | expression '|-|' expression
    | expression SHL expression     // SHL is <<
    | expression '>''>' expression  // check that >> are contiguous
    | expression LE expression      // LE is <=
    | expression GE expression      // GE is >=
    | expression '<' expression
    | expression '>' expression
    | expression NE expression      // NE is !=
    | expression EQ expression      // EQ is ==
```

```
    | expression '&' expression
    | expression '^' expression
    | expression '|' expression
    | expression PP expression       // PP is ++
    | expression AND expression      // AND is &&
    | expression OR expression       // OR is ||
    | expression '?' expression ':' expression
    | expression '<' realTypeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
    | namedType '(' argumentList ')'
    | '(' typeRef ')' expression
    ;

expressionList
    : /* empty */
    | expression
    | expressionList "," expression
    ;

member
    : name
    ;

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList "," argument
    ;

argument
    : expression
    ;

typeArg
    : typeRef
    | nonTypeName
    | VOID
    | "_"
    ;

typeArgumentList
    : /* empty */
```

```
    | typeArg
    | typeArgumentList "," typeArg
    ;
```

See Appendix G for the complete P4 grammar.

This grammar does not indicate the precedence of the various operators. The precedence mostly follows the C precedence rules, with one change and some additions. The precedence of the bitwise operators `&` `|` and `^` is higher than the precedence of relation operators `<`, `<=`, `>`, `>=`. This is more natural given the addition of a true boolean type in the type system, as bitwise operators cannot be applied to boolean types. Concatenation (`++`) has the same precedence as infix addition. Bit-slicing `a[m:l]` has the same precedence as array indexing (`a[i]`).

In addition to these expressions, P4 also supports **`select`** expressions (described in Section 13.6), which may be used only in parsers.

8.1. Expression evaluation order

Given a compound expression, the order in which sub-expressions are evaluated is important when the sub-expressions have side-effects. P4 expressions are evaluated as follows:

- Boolean operators `&&` and `||` use short-circuit evaluation—i.e., the second operand is only evaluated if necessary.
- The conditional operator `e1 ? e2 : e3` evaluates `e1`, and then either evaluates `e2` or `e3`.
- All other expressions are evaluated left-to-right as they appear in the source program.
- Method and function calls are evaluated as described in Section 6.8.

8.2. Operations on `error` types

Symbolic names declared by an **`error`** declaration belong to the **`error`** namespace. The **`error`** type only supports equality (`==`) and inequality (`!=`) comparisons. The result of such a comparison is a Boolean value.

For example, the following operation tests for the occurrence of an error:

```
error errorFromParser;

if (errorFromParser != error.NoError) { /* code omitted */ }
```

8.3. Operations on `enum` types

Symbolic names declared by an enum belong to the namespace introduced by the enum declaration rather than the top-level namespace.

```
enum X { v1, v2, v3 }
X.v1  // reference to v1
v1    // error - v1 is not in the top-level namespace
```

Similar to errors, **`enum`** expressions without a specified underlying type only support equality (`==`) and inequality (`!=`) comparisons. Expressions whose type is an **`enum`** without a specified underlying type

cannot be cast to or from any other type.

An `enum` may also specify an underlying type, such as the following:

```
enum bit<8> E {
  e1 = 0,
  e2 = 1,
  e3 = 2
}
```

More than one symbolic value in an `enum` may map to the same fixed-width integer value.

```
enum bit<8> NonUnique {
  b1 = 0,
  b2 = 1,  // Note, both b2 and b3 map to the same value.
  b3 = 1,
  b4 = 2
}
```

An `enum` with an underlying type also supports explicit casts to and from the underlying type. For instance, the following code:

```
bit<8> x;
E a = E.e2;
E b;

x = (bit<8>) a; // sets x to 1
b = (E) x;      // sets b to E.e2
```

... casts `a` (which was initialized with `E.e2`) to a `bit<8>`, using the specified fixed-width unsigned integer representation for `E.e2`, i.e. `1`. The variable `b` is then set to the symbolic value `E.e2`, which corresponds to the fixed-width unsigned integer value `1`.

Because it is always safe to cast from an `enum` to its underlying fixed-width integer type, implicit casting from an `enum` to its fixed-width (signed or unsigned) integer type is also supported (see Section 8.11.2):

```
bit<8> x = E.e2; // sets x to 1 (E.e2 is automatically casted to bit<8>)

E  a = E.e2
bit<8> y = a << 3; // sets y to 8 (a is automatically casted to bit<8> and then shifted)
```

Implicit casting from an underlying fixed-width type to an enum is not supported.

```
enum bit<8> E1 {
   e1 = 0, e2 = 1, e3 = 2
}

enum bit<8> E2 {
```

```
   e1 = 10, e2 = 11, e3 = 12
}
E1 a = E1.e1;
E2 b = E2.e2;

a = b;      // Error: b is automatically casted to bit<8>,
            // but bit<8> cannot be automatically casted to E1

a = (E1) b; // OK

a = E1.e1 + 1; // Error: E.e1 is automatically casted to bit<8>,
               // and the right-hand expression has
               // the type bit<8>, which cannot be casted to E automatically.

a = (E1)(E1.e1 + 1); // Final explicit casting makes the assignment legal

a = E1.e1 + E1.e2; // Error: both arguments to the addition are automatically
                   // casted to bit<8>. Thus the addition itself is legal, but
                   // the assignment is not

a = (E1)(E2.e1 + E2.e2); //  Final explicit casting makes the assignment legal
```

A reasonable compiler might generate a warning in cases that involve multiple automatic casts.

```
E1     a = E1.e1;
E2     b = E2.e2;
bit<8> c;

if (a > b) { // Potential warning: two automatic and different casts to bit<8>.
   // code omitted
}

c = a + b; // Legal, but a warning would be reasonable
```

Note that while it is always safe to cast from an `enum` to its fixed-width unsigned integer type, and vice versa, there may be cases where casting a fixed-width unsigned integer value to its related `enum` type produces an unnamed value.

```
bit<8> x = 5;
E e = (E) x; // sets e to an unnamed value
```

sets `e` to an unnamed value, since there is no symbol corresponding to the fixed-width unsigned integer value `5`.

For example, in the following code, the `else` clause of the `if`/`else if`/`else` block can be reached even though the matches on `x` are complete with respect to the symbols defined in `MyPartialEnum_t`:

```
enum bit<2> MyPartialEnum_t {
    VALUE_A = 2w0,
    VALUE_B = 2w1,
    VALUE_C = 2w2
}

bit<2> y = < some value >;
MyPartialEnum_t x = (MyPartialEnum_t)y;

if (x == MyPartialEnum_t.VALUE_A) {
    // some code here
} else if (x == MyPartialEnum_t.VALUE_B) {
    // some code here
} else if (x == MyPartialEnum_t.VALUE_C) {
    // some code here
} else {
    // A P4 compiler MUST ASSUME that this branch can be executed
    // some code here
}
```

Additionally, if an enumeration is used as a field of a header, we would expect the **transition select** to match **default** when the parsed integer value does not match one of the symbolic values of `EtherType` in the following example:

```
enum bit<16> EtherType {
  VLAN      = 0x8100,
  IPV4      = 0x0800,
  IPV6      = 0x86dd
}

header ethernet {
  // Some fields omitted
  EtherType etherType;
}

parser my_parser(/* parameters omitted */) {
  state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
      EtherType.VLAN : parse_vlan;
      EtherType.IPV4 : parse_ipv4;
      EtherType.IPV6: parse_ipv6;
      default: reject;
  }
}
```

Any variable with an `enum` type that contains an unnamed value (whether as the result of a cast to an `enum` with an underlying type, parse into the field of an `enum` with an underlying type, or simply the declaration of any `enum` without a specified initial value) will not be equal to any of the values defined for that type. Such an unnamed value should still lead to predictable behavior in cases where any legal value would match, e.g. it should match in any of these situations:

- If used in a `select` expression, it should match `default` or _ in a key set expression.
- If used as a key with `match_kind` `ternary` in a table, it should match a table entry where the field has all bit positions "don't care".
- If used as a key with `match_kind` `lpm` in a table, it should match a table entry where the field has a prefix length of 0.

Note that if an `enum` value lacking an underlying type appears in the control-plane API, the compiler must select a suitable serialization data type and representation. For `enum` values with an underlying type and representations, the compiler should use the specified underlying type as the serialization data type and representation.

Additionally, the size of a serializable enum can be determined at compile-time. However, the size of an enum without an underlying type cannot be determined at compile-time (Section 9).

8.4. Operations on `match_kind` types

Values of type `match_kind` are similar to `enum` values. They support only assignment and comparisons for equality and inequality.

```
match_kind { fuzzy }
const bool same = exact == fuzzy;  // always 'false'
```

8.5. Expressions on Booleans

The following operations are provided on Boolean expressions: - "And", denoted by `&&` - "Or", denoted by `||` - Negation, denoted by `!` - Equality and inequality tests, denoted by `==` and `!=` respectively.

The precedence of these operators is similar to C and uses short-circuited evaluation where relevant.

Additionally, the size of a boolean can be determined at compile-time (Section 9).

P4 does not implicitly cast from bit-strings to Booleans or vice versa. As a consequence, a program that is valid in a language like C such as,

```
if (x) /* body omitted */
```

(where x has an integer type) must instead be written in P4 as:

```
if (x != 0) /* body omitted */
```

See the discussion on arbitrary-precision types and implicit casts in Section 8.11.2 for details on how the `0` in this expression is evaluated.

8.5.1. Conditional operator

A conditional expression of the form `e1 ? e2 : e3` behaves the same as in languages like C. As described above, the expression `e1` is evaluated first, and second either `e2` or `e3` is evaluated depending on the result.

The first sub-expression `e1` must have Boolean type and the second and third sub-expressions must have the same type, which cannot both be arbitrary-precision integers unless the condition itself can be evaluated at compilation time. This restriction is designed to ensure that the width of the result of the conditional expression can be inferred statically at compile time.

8.6. Operations on fixed-width bit types (unsigned integers)

This section discusses all operations that can be performed on expressions of type `bit<W>` for some width `W`, also known as bit-strings.

Arithmetic operations "wrap around", similar to C operations on unsigned values (i.e., representing a large value on W bits will only keep the least-significant W bits of the value). In particular, P4 does not have arithmetic exceptions—the result of an arithmetic operation is defined for all possible inputs.

P4 target architectures may optionally support saturating arithmetic. All saturating operations are limited to a fixed range between a minimum and maximum value. Saturating arithmetic has advantages, in particular when used as counters. The result of a saturating counter max-ing out is much closer to the real result than a counter that overflows and wraps around. According to Wikipedia Saturating Arithmetic saturating arithmetic is as numerically close to the true answer as possible; for 8-bit binary signed arithmetic, when the correct answer is 130, it is considerably less surprising to get an answer of 127 from saturating arithmetic than to get an answer of –126 from modular arithmetic. Likewise, for 8-bit binary unsigned arithmetic, when the correct answer is 258, it is less surprising to get an answer of 255 from saturating arithmetic than to get an answer of 2 from modular arithmetic. At this time, P4 defines saturating operations only for addition and subtraction. For an unsigned integer with bit-width of `W`, the minimum value is `0` and the maximum value is `2^W-1`. The precedence of saturating addition and subtraction operations is the same as for modular arithmetic addition and subtraction.

All binary operations except shifts and concatenation require both operands to have the same exact type and width; supplying operands with different widths produces an error at compile time. No implicit casts are inserted by the compiler to equalize the widths. There are no other binary operations that accept signed and unsigned values simultaneously besides shifts and concatenation. The following operations are provided on bit-string expressions:

- Test for equality between bit-strings of the same width, designated by `==`. The result is a Boolean value.
- Test for inequality between bit-strings of the same width, designated by `!=`. The result is a Boolean value.
- Unsigned comparisons `<,>,<=,>=`. Both operands must have the same width and the result is a Boolean value.

Each of the following operations produces a bit-string result when applied to bit-strings of the same width:

- Negation, denoted by unary `-`. The result is computed by subtracting the value from 2^W. The result is unsigned and has the same width as the input. The semantics is the same as the C negation of unsigned numbers.

- Unary plus, denoted by `+`. This operation behaves like a no-op.
- Addition, denoted by `+`. This operation is associative. The result is computed by truncating the result of the addition to the width of the output (similar to C).
- Subtraction, denoted by `-`. The result is unsigned, and has the same type as the operands. It is computed by adding the negation of the second operand (similar to C).
- Multiplication, denoted by `*`. The result has the same width as the operands and is computed by truncating the result to the output's width. P4 architectures may impose additional restrictions — e.g., they may only allow multiplication by a non-negative integer power of two.
- Bitwise "and" between two bit-strings of the same width, denoted by `&`.
- Bitwise "or" between two bit-strings of the same width, denoted by `|`.
- Bitwise "complement" of a single bit-string, denoted by `~`.
- Bitwise "xor" of two bit-strings of the same width, denoted by `^`.
- Saturating addition, denoted by `|+|`.
- Saturating subtraction, denoted by `|-|`.

Bit-strings also support the following operations:

- Logical shift left and right by a non-negative integer value (which need not be a compile-time known value), denoted by `<<` and `>>` respectively. In a shift, the left operand is unsigned, and right operand must be either an expression of type `bit<S>` or a non-negative integer value that is known at compile time. The result has the same type as the left operand. Shifting by an amount greater than or equal to the width of the input produces a result where all bits are zero.
- Extraction of a set of contiguous bits, also known as a slice, denoted by `[H:L]`, where `H` and `L` must be expressions that evaluate to non-negative, local compile-time known values, and `H >= L`. The types of `H` and `L` (which do not need to be identical) must be numeric (Section 7.4). The result is a bit-string of width `H - L + 1`, including the bits numbered from `L` (which becomes the least significant bit of the result) to `H` (the most significant bit of the result) from the source operand. The conditions `0 <= L <= H < W` are checked statically (where `W` is the length of the source bit-string). Note that both endpoints of the extraction are inclusive. The bounds are required to be local compile-time known values so that the width of the result can be computed at compile time. Slices are also l-values, which means that P4 supports assigning to a slice: `e[H:L] = x` . The effect of this statement is to set bits `H` through `L` (inclusive of both) of `e` to the bit-pattern represented by `x`, and leaves all other bits of `e` unchanged. A slice of an unsigned integer is an unsigned integer.
- Concatenation of bit-strings and/or fixed-width signed integers, denoted by `++`. The two operands must be either `bit<W>` or `int<W>`, and they can be of different signedness and width. The result has the same signedness as the left operand and the width equal to the sum of the two operands' width. In concatenation, the left operand is placed as the most significant bits.

Additionally, the size of a bit-string can be determined at compile-time (Section 9).

8.7. Operations on fixed-width signed integers

This section discusses all operations that can be performed on expressions of type `int<W>` for some `W`. Recall that the `int<W>` denotes signed `W`-bit integers, represented using two's complement.

In general, P4 arithmetic operations do not detect "underflow" or "overflow": operations simply "wrap around", similar to C operations on unsigned values. Hence, attempting to represent large values using `W` bits will only keep the least-significant `W` bits of the value.

P4 supports saturating arithmetic (addition and subtraction) for signed integers. Targets may optionally reject programs using saturating arithmetic. For a signed integer with bit-width of `W`, the minimum value is `-2^(W-1)` and the maximum value is `2^(W-1)-1`.

P4 also does not support arithmetic exceptions. The runtime result of an arithmetic operation is defined for all combinations of input arguments.

All binary operations except shifts and concatenation require both operands to have the same exact type (signedness) and width and supplying operands with different widths or signedness produces a compile-time error. No implicit casts are inserted by the compiler to equalize the types. Except for shifts and concatenation, P4 does not have any binary operations that operate simultaneously on signed and unsigned values.

Note that bitwise operations on signed integers are well-defined, since the representation is mandated to be two's complement.

The `int<W>` datatype supports the following operations; all binary operations require both operands to have the exact same type. The result always has the same width as the left operand.

- Negation, denoted by unary `-`.
- Unary plus, denoted by `+`. This operation behaves like a no-op.
- Addition, denoted by `+`.
- Subtraction, denoted by `-`.
- Comparison for equality and inequality, denoted `==` and `!=` respectively. These operations produce a Boolean result.
- Numeric comparisons, denoted by `<`, `<=`, `>`, and `>=`. These operations produce a Boolean result.
- Multiplication, denoted by `*`. Result has the same width as the operands. P4 architectures may impose additional restrictions—e.g., they may only allow multiplication by a power of two.
- Bitwise "and" between two bit-strings of the same width, denoted by `&`.
- Bitwise "or" between two bit-strings of the same width, denoted by `|`.
- Bitwise "complement" of a single bit-string, denoted by `~`.
- Bitwise "xor" of two bit-strings of the same width, denoted by `^`.
- Saturating addition, denoted by `|+|`.
- Saturating subtraction, denoted by `|-|`.

The `int<W>` datatype also support the following operations:

- Arithmetic shift left and right denoted by `<<` and `>>`. The left operand is signed and the right operand must be either an unsigned number of type `bit<S>` or a compile-time known value that is a non-negative integer. The result has the same type as the left operand. Shifting left produces the exact same bit pattern as a shift left of an unsigned value. Shift left can thus overflow, when it leads to a change of the sign bit. Shifting by an amount greater than the width of the input produces a "correct" result:
  - all result bits are zero when shifting left
  - all result bits are zero when shifting a non-negative value right
  - all result bits are one when shifting a negative value right
- Extraction of a set of contiguous bits, also known as a slice, denoted by `[H:L]`, where `H` and `L` must be expressions that evaluate to non-negative, local compile-time known values, and `H >= L` must be true. The types of `H` and `L` (which do not need to be identical) must be numeric (Section 7.4). The result is an unsigned bit-string of width `H - L + 1`, including the bits numbered from `L` (which

becomes the least significant bit of the result) to H (the most significant bit of the result) from the source operand. The conditions 0 <= L <= H < W are checked statically (where W is the length of the source bit-string). Note that both endpoints of the extraction are inclusive. The bounds are required to be values that are known at compile time so that the width of the result can be computed at compile time. Slices are also l-values, which means that P4 supports assigning to a slice: e[H:L] = x . The effect of this statement is to set bits H through L of e to the bit-pattern represented by x, and leaves all other bits of e unchanged. A slice of a signed integer is treated as an unsigned integer.

- Concatenation of bit-strings and/or fixed-width signed integers, denoted by ++. The two operands must be either **bit**<W> or **int**<W>, and they can be of different signedness and width. The result has the same signedness as the left operand and the width equal to the sum of the two operands' width. In concatenation, the left operand is placed as the most significant bits.

Additionally, the size of a fixed-width signed integer can be determined at compile-time (Section 9).

8.8. Operations on arbitrary-precision integers

The type **int** denotes arbitrary-precision integers. In P4, all expressions of type **int** must be compile-time known values. The type **int** supports the following operations:

- Negation, denoted by unary –
- Unary plus, denoted by +. This operation behaves like a no-op.
- Addition, denoted by +.
- Subtraction, denoted by –.
- Comparison for equality and inequality, denoted by == and != respectively. These operations produce a Boolean result.
- Numeric comparisons <, <=, >, and >=. These operations produce a Boolean result.
- Multiplication, denoted by *.
- Truncating integer division between positive values, denoted by /.
- Modulo between positive values, denoted by %.
- Arithmetic shift left and right denoted by << and >>. These operations produce an **int** result. The right operand must be either an unsigned value of type **bit**<S> or a compile-time known value that is a non-negative integer. The expression a << b is equal to $a \times 2^b$ while a >> b is equal to $\lfloor a/2^b \rfloor$.
- Bit slices, denoted by [H:L], where H and L must be expressions that evaluate to non-negative, local compile-time known values, and H >= L must be true. The types of H and L (which do not need to be identical) must be one of the following:
  - **int** - an arbitrary-precision integer (section 7.1.6.5)
  - **bit**<W> - a W-bit unsigned integer where W >= 0 (section 7.1.6.2)
  - **int**<W> - a W-bit signed integer where W >= 1 (section 7.1.6.3)

– a serializable **enum** with an underlying type that is **bit**<W> or **int**<W> (section 7.2.1).

The result is an unsigned bit-string of width `H - L + 1`, including the bits numbered from `L` (which becomes the least significant bit of the result) to `H` (the most significant bit of the result) from the source operand. The conditions `0 <= L <= H` are checked statically. If necessary, the source integer value that is sliced is automatically extended to have a width with `H` bits. Note that both endpoints of the extraction are inclusive. The bounds are required to be values that are known at compile time so that the width of the result can be computed at compile time. A slice of a negative or positive value is always a positive value.

Each operand that participates in any of these operation must have type **int** (except shifts). Binary operations cannot be used to combine values of type **int** with values of a fixed-width type (except shifts). However, the compiler automatically inserts casts from **int** to fixed-width types in certain situations—see Section 8.11.

All computations on **int** values are carried out without loss of information. For example, multiplying two 1024-bit values may produce a 2048-bit value (note that concrete representation of **int** values is not specified). **int** values can be cast to **bit**<w> and **int**<w> values. Casting an **int** value to a fixed-width type will preserve the least-significant bits. If truncation causes significant bits to be lost, the compiler should emit a warning.

Note: bitwise-operations (`|`,`&`,`^`,`~`) are not defined on expressions of type **int**. In addition, it is illegal to apply division and modulo to negative values.

Note: saturating arithmetic is not supported for arbitrary-precision integers.

8.9. Concatenation and shifts

8.9.1. Concatenation

Concatenation is applied to two bit-strings (signed or unsigned). It is denoted by the infix operator ++. The result is a bit-string whose length is the sum of the lengths of the inputs where the most significant bits are taken from the left operand; the sign of the result is taken from the left operand.

8.9.2. A note about shifts

The left operand of shifts can be any one out of unsigned bit-strings, signed bit-strings, and arbitrary-precision integers, and the right operand of shifts must be either an expression of type **bit**<S> or a compile-time known value that is a non-negative integer. The result has the same type as the left operand.

Shifts on signed and unsigned bit-strings deserve a special discussion for the following reasons:

- Right shift behaves differently for signed and unsigned bit-strings: right shift for signed bit-strings is an arithmetic shift, and for unsigned bit-strings is a logical shift.
- Shifting with a negative amount does not have a clear semantics: the P4 type system makes it illegal to shift with a negative amount.
- Unlike C, shifting by an amount larger than or equal to the number of bits has a well-defined result.
- Finally, depending on the capabilities of the target, shifting may require doing work which is exponential in the number of bits of the right-hand-side operand.

Consider the following examples:

```
bit<8> x;
bit<16> y;
bit<16> z = y << x;
bit<16> w = y << 1024;
```

As mentioned above, P4 gives a precise meaning shifting with an amount larger than the size of the shifted value, unlike C.

P4 targets may impose additional restrictions on shift operations such as forbidding shifts by non-constant expressions, or by expressions whose width exceeds a certain bound. For example, a target may forbid shifting an 8-bit value by a non-constant value whose width is greater than 3 bits.

8.10. Operations on variable-size bit types

To support parsing headers with variable-length fields, P4 offers a type **varbit**. Each occurrence of the type **varbit** has a statically-declared maximum width, as well as a dynamic width, which must not exceed the static bound. Prior to initialization a variable-size bit-string has an unknown dynamic width.

Variable-length bit-strings support a limited set of operations:

- Assignment to another variable-sized bit-string. The target of the assignment must have the same static width as the source. When executed, the assignment sets the dynamic width of the target to the dynamic width of the source.
- Comparison for equality or inequality with another **varbit** field. Two **varbit** fields can be compared only if they have the same type. Two varbits are equal if they have the same dynamic width and all the bits up to the dynamic width are the same.

The following operations are not supported directly on a value of type **varbit**, but instead on any type for which `extract` and `emit` operations are supported (e.g. a value with type header) that may contain a field of type **varbit**. They are mentioned here only to ease finding this information in a section dedicated to type **varbit**.

- Parser extraction into a header containing a variable-sized bit-string using the two-argument `extract` method of a `packet_in` extern object (see Section 13.8.2). This operation sets the dynamic width of the field.
- The `emit` method of a `packet_out` extern object can be performed on a header and a few other types (see Section 16) that contain a field with type **varbit**. Such an `emit` method call inserts a variable-sized bit-string with a known dynamic width into the packet being constructed.

Additionally, the maximum size of a variable-length bit-string can be determined at compile-time (Section 9).

8.11. Casts

P4 provides a limited set of casts between types. A cast is written `(t) e`, where `t` is a type and `e` is an expression. Casts are only permitted on base types and derived types introduced by **typedef**, **type**, and **enum**. While this design is arguably more onerous for programmers, it has several benefits:

- It makes user intent unambiguous.

- It makes the costs associated with converting numeric values explicit. Implementing certain casts involves sign extensions, and thus can require significant computational resources on some targets.
- It reduces the number of cases that have to be considered in the P4 specification. Some targets may not support all casts.

8.11.1. Explicit casts

The following casts are legal in P4:

- `bit<1>` ↔ `bool`: converts the value `0` to `false`, the value `1` to `true`, and vice versa.
- `int` → `bool`: only if the `int` value is `0` (converted to `false`) or `1` (converted to `true`)
- `int<W>` → `bit<W>`: preserves all bits unchanged and reinterprets negative values as positive values
- `bit<W>` → `int<W>`: preserves all bits unchanged and reinterprets values whose most-significant bit is `1` as negative values
- `bit<W>` → `bit<X>`: truncates the value if `W > X`, and otherwise (i.e., if `W <= X`) pads the value with zero bits.
- `int<W>` → `int<X>`: truncates the value if `W > X`, and otherwise (i.e., if `W < X`) extends it with the sign bit.
- `bit<W>` → `int`: preserves the value unchanged but converts it to an unlimited-precision integer; the result is always non-negative
- `int<W>` → `int`: preserves the value unchanged but converts it to an unlimited-precision integer; the result may be negative
- `int` → `bit<W>`: converts the integer value into a sufficiently large two's complement bit string to avoid information loss, and then truncates the result to `W` bits. The compiler should emit a warning on overflow or on conversion of negative value.
- `int` → `int<W>`: converts the integer value into a sufficiently-large two's complement bit string to avoid information loss, and then truncates the result to `W` bits. The compiler should emit a warning on overflow.
- casts between two types that are introduced by `typedef` and are equivalent to one of the above combinations.
- casts between a typedef and the original type.
- casts between a type introduced by `type` and the original type.
- casts between an `enum` with an explicit type and its underlying type
- casts of a key-value list to a struct type or a header type (see Section 8.13)
- casts of a tuple expression to a header stack type
- casts of an invalid expression `{#}` to a header or a header union type
- casts where the destination type is the same as the source type if the destination type appears in this list (this excludes e.g., parsers or externs).

8.11.2. Implicit casts

To keep the language simple and avoid introducing hidden costs, P4 only implicitly casts from `int` to fixed-width types and from enums with an underlying type to the underlying type. In particular, applying a binary operation (except shifts and concatenation) to an expression of type `int` and an expression with a fixed-width type will implicitly cast the `int` expression to the type of the other expression. For enums with an underlying type, it can be implicitly cast to its underlying type whenever appropriate,

including but not limited to in shifts, concatenation, bit slicing indexes, header stack indexes as well as other unary and binary operations.

For example, given the following declarations,

```
enum bit<8> E {
   a = 5
}

bit<8>  x;
bit<16> y;
int<8>  z;
```

the compiler will add implicit casts as follows:

- `x + 1` becomes `x + (bit<8>)1`
- `z < 0` becomes `z < (int<8>)0`
- `x | 0xFFF` becomes `x | (bit<8>)0xFFF`; overflow warning
- `x + E.a` becomes `x + (bit<8>)E.a`
- `x &&& 8` becomes `x &&& (bit<8>)8`
- `x << 256` remains unchanged; `256` not implicitly cast to `8w0` in a shift; overflow warning
- `16w11 << E.a` becomes `16w11 << (bit<8>)E.a`
- `x[E.a:0]` becomes `x[(bit<8>)E.a:0]`
- `E.a ++ 8w0` becomes `(bit<8>)E.a ++ 8w0`

The compiler also adds implicit casts when types of different expressions need to match‘’; for example, as described in Section 13.6, since select labels are compared against the selected expression, the compiler will insert implicit casts for the select labels when they have `int` types. Similarly, when assigning a structure-valued expression to a structure or header, the compiler will add implicit casts for `int` fields.

8.11.3. Illegal arithmetic expressions

Many arithmetic expressions that would be allowed in other languages are illegal in P4. To illustrate, consider the following declarations:

```
bit<8>  x;
bit<16> y;
int<8>  z;
```

The table below shows several expressions which are illegal because they do not obey the P4 typing rules. For each expression we provide several ways that the expression could be manually rewritten into a legal expression. Note that for some expression there are several legal alternatives, which may produce different results! The compiler cannot guess the user intent, so P4 requires the user to disambiguate.

Expression	Why it is illegal	Alternatives
`x + y`	Different widths	`(bit<16>)x + y` `x + (bit<8>)y`

`x + z`	Different signedness	`(int<8>)x + z` `x + (bit<8>)z`
`(int<8>)y`	Cannot change both sign and width	`(int<8>)(bit<8>)y` `(int<8>)(int<16>)y`

`y + z`	Different widths and signs	`(int<8>)(bit<8>)y + z`
		`y + (bit<16>)(bit<8>)z`
		`(bit<8>)y + (bit<8>)z`
		`(int<16>)y + (int<16>)z`

`x << z`	RHS of shift cannot be signed	`x << (bit<8>)z`
`x < z`	Different signs	`x < (bit<8>)z` `(int<8>)x < z`
`1 << x`	Either LHS should have a fixed width (bit shift), Or RHS must be compile-time known (int shift)	`32w1 << x` None
`~1`	Bitwise operation on int	`~32w1`
`5 & -3`	Bitwise operation on int	`32w5 & -3`

8.12. Operations on tuple expressions

Tuples can be assigned to other tuples with the same type, passed as arguments and returned from functions, and can be initialized with tuple expressions.

```
tuple<bit<32>, bool> x = { 10, false };
```

The fields of a tuple can be accessed using array index syntax `x[0]`, `x[1]`. The indexes must be local compile-time known values, to enable the type-checker to identify the field types statically.

Tuples can be compared for equality using `==` and `!=`; two tuples are equal if and only if all their fields are respectively equal.

Currently tuple fields are not left-values, even if the tuple itself is. (I.e. a tuple can only be assigned monolithically, and the field values cannot be changed individually.) This restriction may be lifted in a future version of the language.

A tuple expression is written using curly braces, with each element separated by a comma:

```
expression ...
    | '{' expressionList '}'

expressionList
    : /* empty */
    | expression
    | expressionList "," expression
    ;
```

The type of a tuple expression is a tuple type (Section 7.2.6). Tuple expressions can be assigned to expressions of type **`tuple`**, **`struct`** or **`header`**, and can also be passed as arguments to methods. Tuples may be nested. However, tuple expressions are not l-values.

For example, the following program fragment uses a tuple expression to pass several header fields simultaneously to a learning provider:

```
extern LearningProvider<T> {
    LearningProvider();
    void learn(in T data);
}

LearningProvider<tuple<bit<48>, bit<32>>>() lp;

lp.learn( { hdr.ethernet.srcAddr, hdr.ipv4.src } );
```

A tuple may be used to initialize a structure if the tuple has the same number of elements as fields in the structure. The effect of such an initializer is to assign the n^{th} element of the tuple to the n^{th} field in the structure:

```
struct S {
    bit<32> a;
    bit<32> b;
}
const S x = { 10, 20 }; //a = 10, b = 20
```

A tuple expression can have an explicit structure or header type specified, and then it is converted automatically to a structure-valued expression (see 8.13):

```
struct S {
    bit<32> a;
    bit<32> b;
}

extern void f<T>(in T data);

f((S){ 10, 20 }); // automatically converted to f((S){a = 10, b = 20});
```

Tuple expressions can also be used to initialize variables whose type is a `tuple` type.

```
tuple<bit<32>, bool> x = { 10, false };
```

The empty tuple expression has type `tuple`<> - a tuple with no components.

8.13. Operations on structure-valued expressions

One can write expressions that evaluate to a structure or header. The syntax of these expressions is given by:

```
expression ...
    | '{' kvList '}'
    | '(' typeRef ')' expression
    ;
```

```
kvList
    : kvPair
    | kvList "," kvPair
    ;

kvPair
    : name "=" expression
    ;
```

For a structure-valued expression `typeRef` is the name of a **`struct`** or **`header`** type. The `typeRef` can be omitted if it can be inferred from context, e.g., when initializing a variable with a **`struct`** type. Structure-valued expressions that evaluate to a value of some **`header`** type are always valid.

The following example shows a structure-valued expression used in an equality comparison expression:

```
struct S {
    bit<32> a;
    bit<32> b;
}

S s;

// Compare s with a structure-valued expression
bool b = s == (S) { a = 1, b = 2 };
```

Structure-valued expressions can be used in the right-hand side of assignments, in comparisons, in field selection expressions, and as arguments to functions, method or actions. Structure-valued expressions are not left values.

Structure-valued expressions that do not have `...` as their last element must provide a value for every member of the struct or header type to which it evaluates, by mentioning each field name exactly once.

Structure-valued expressions that have `...` as their last element are allowed to give values to only a subset of the fields of the struct or header type to which it evaluates. Any field names not given a value explicitly will be given their default value (see Section 8.26).

The order of the fields of the **`struct`** or **`header`** type does not need to match the order of the values of the structure-valued expression.

It is a compile-time error if a field name appears more than once in the same structure-valued expression.

8.14. Operations on lists

The value of a list is written using curly braces, with each element separated by a comma. The left curly brace is preceded by a `(list<T>)` where `T` is the list element type. Such a value can be passed as an argument, e.g. to extern constructor functions.

```
struct pair_t {
    bit<16> a;
    bit<32> b;
}

extern E {
    E(list<pair_t> data);
    void run();
}

control c() {
    E((list<pair_t>) {{2, 3}, {4, 5}}) e;
    apply {
        e.run();
    }
}
```

Additionally, the size of a list can be determined at compile-time (Section 9).

8.15. Operations on sets

Some P4 expressions denote sets of values (`set<T>`, for some type `T`; see Section 7.2.9.1). These expressions can appear only in a few contexts—parsers and table entries. For example, the **`select`** expression (Section 13.6) has the following structure:

```
select (expression) {
   set1: state1;
   set2: state2;
   // More labels omitted
}
```

Here the expressions `set1`, `set2`, etc. evaluate to sets of values and the **`select`** expression tests whether `expression` belongs to the sets used as labels.

```
keysetExpression
    : tupleKeysetExpression
    | simpleKeysetExpression
    ;

tupleKeysetExpression
    : "(" simpleKeysetExpression "," simpleExpressionList ")"
    | "(" reducedSimpleKeysetExpression ")"
    ;

simpleExpressionList
    : simpleKeysetExpression
```

```
    | simpleExpressionList "," simpleKeysetExpression
    ;

reducedSimpleKeysetExpression
    : expression "&&&" expression
    | expression ".." expression
    | DEFAULT
    | "_"
    ;

simpleKeysetExpression
    : expression
    | expression "&&&" expression
    | expression ".." expression
    | DEFAULT
    | "_"
    ;
```

The mask (`&&&`) and range (`..`) operators have the same precedence; the just above the `?:` operator.

8.15.1. Singleton sets

In a set context, expressions denote singleton sets. For example, in the following program fragment,

```
select (hdr.ipv4.version) {
   4: continue;
}
```

The label `4` denotes the singleton set containing the `int` value `4`.

8.15.2. The universal set

In a set context, the expressions `default` or `_` denote the universal set, which contains all possible values of a given type:

```
select (hdr.ipv4.version) {
   4: continue;
   _: reject;
}
```

8.15.3. Masks

The infix operator `&&&` takes two arguments of the same numeric type (Section 7.4), and creates a value of the same type. The right value is used as a "mask", where each bit set to `0` in the mask indicates a "don't care" bit. More formally, the set denoted by `a &&& b` is defined as follows:

```
a &&& b = { c where a & b = c & b }
```

For example:

```
8w0x0A &&& 8w0x0F
```

denotes a set that contains 16 different `bit<8>` values, whose bit-pattern is `XXXX1010`, where the value of an `X` can be any bit. Note that there may be multiple ways to express a keyset using a mask operator—e.g., `8w0xFA &&& 8w0x0F` denotes the same keyset as in the example above.

Similar to other binary operations, the mask operator allows the compiler to automatically insert casts to unify the argument types in certain situations (section 8.11.2).

P4 architectures may impose additional restrictions on the expressions on the left and right-hand side of a mask operator: for example, they may require that either or both sub-expressions be compile-time known values.

8.15.4. Ranges

The infix operator `..` takes two arguments of the same numeric type `T` (Section 7.4), and creates a value of the type `set<T>`. The set contains all values numerically between the first and the second, inclusively. For example:

```
4s5 .. 4s8
```

denotes a set of 4 consecutive `int<4>` values `4s5`, `4s6`, `4s7`, and `4s8`.

Similar to other binary operations, the range operator allows the compiler to automatically insert casts to unify the argument types in certain situations (section 8.11.2).

A range where the second value is smaller than the first one represents an empty set.

8.15.5. Products

Multiple sets can be combined using Cartesian product:

```
select(hdr.ipv4.ihl, hdr.ipv4.protocol) {
     (4w0x5, 8w0x1): parse_icmp;
     (4w0x5, 8w0x6): parse_tcp;
     (4w0x5, 8w0x11): parse_udp;
     (_, _): accept; }
```

The type of a product of sets is a set of tuples.

8.16. Operations on struct types

The only operation defined on expressions whose type is a **struct** is field access, written using dot (".") notation—e.g., `s.field`. If `s` is an l-value, then `s.field` is also an l-value. P4 also allows copying **struct**s using assignment when the source and target of the assignment have the same type. Finally, **struct**s can be initialized with a tuple expression, as discussed in Section 8.12, or with a structure-valued expression, as described in 8.13. Both of these cases must initialize all fields of the structure. The size of a struct can be determined at compile-time (Section 9).

Two structs can be compared for equality (==) or inequality (!=) only if they have the same type and all of their fields can be recursively compared for equality. Two structures are equal if and only if all their corresponding fields are equal.

The following example shows a structure initialized in several different ways:

```
struct S {
    bit<32> a;
    bit<32> b;
}
const S x = { 10, 20 };             // tuple expression
const S x = { a = 10, b = 20 };     // structure-valued expression
const S x = (S) { a = 10, b = 20 }; // structure-valued expression
```

See Section 8.25 for a description of the behavior if **struct** fields are read without being initialized.

8.17. Operations on headers

Headers provide the same operations as **struct**s. Assignment between headers also copies the "validity" header bit.

In addition, headers support the following methods:

- The method `isValid()` returns the value of the "validity" bit of the header.
- The method `setValid()` sets the header's validity bit to "true". It can only be applied to an l-value.
- The method `setInvalid()` sets the header's validity bit to "false". It can only be applied to an l-value.

Similar to a **struct**, a header object can be initialized with a tuple expression (see Section 8.12) — the tuple fields are assigned to the header fields in the order they appear — or with a structure-valued expression (see Section 8.16). When initialized the header automatically becomes valid:

```
header H { bit<32> x; bit<32> y; }
H h;
h = { 10, 12 };  // This also makes the header h valid
h = { y = 12, x = 10 };  // Same effect as above
```

Two headers can be compared for equality (==) or inequality (!=) only if they have the same type. Two headers are equal if and only if they are both invalid, or they are both valid and all their corresponding fields are equal. Furthermore, the size of a header can be determined at compile-time (Section 9).

The expression `{#}` represents an invalid header of some type, but it can be any header or header union type. A P4 compiler may require an explicit cast on this expression in cases where it cannot determine the particular header or header union type from the context.

```
expression
    ...
    | "{#}"
```

For example:

```
header H { bit<32> x; bit<32> y; }
H h;
h = {#};   // This make the header h become invalid
if (h == {#}) {     // This is equivalent to the condition !h.isValid()
    // ...
}
```

Note that the `#` character cannot be misinterpreted as a preprocessor directive, since it cannot be the first character on a line when it occurs in the single lexical token `{#}`, which may not have whitespace or any other characters between those shown.

See Section 8.25 for a description of the behavior if header fields are read without being initialized, or header fields are written to a currently invalid header.

8.18. Operations on header stacks

A header stack is a fixed-size array of headers with the same type. The valid elements of a header stack need not be contiguous. P4 provides a set of computations for manipulating header stacks. A header stack `hs` of type `h[n]` can be understood in terms of the following pseudocode:

```
// type declaration
struct hs_t {
  bit<32> nextIndex;
  bit<32> size;
  h[n] data;  // Ordinary array
}

// instance declaration and initialization
hs_t hs;
hs.nextIndex = 0;
hs.size = n;
```

Intuitively, a header stack can be thought of as a struct containing an ordinary array of headers `hs` and a counter `nextIndex` that can be used to simplify the construction of parsers for header stacks, as discussed below. The `nextIndex` counter is initialized to `0`.

Given a header stack value `hs` of size `n`, the following expressions are legal:

- `hs[index]`: produces a reference to the header at the specified position within the stack; if `hs` is an l-value, the result is also an l-value. The header may be invalid. Some implementations may impose the constraint that the index expression must be a compile-time known value. A P4 compiler must give an error if an index that is a compile-time known value is out of range.

  Accessing a header stack `hs` with an index less than `0` or greater than or equal to `hs.size` results in an undefined value. See Section 8.25 for more details.

  The `index` is an expression that must be of numeric types (Section 7.4).

- `hs.size`: produces a 32-bit unsigned integer that returns the size of the header stack (a local compile-time known value).

- assignment from a header stack `hs` into another stack requires the stacks to have the same types and sizes. All components of `hs` are copied, including its elements and their validity bits, as well as `nextIndex`.

To help programmers write parsers for header stacks, P4 also offers computations that automatically advance through the stack as elements are parsed:

- `hs.next`: produces a reference to the element with index `hs.nextIndex` in the stack. May only be used in a **`parser`**. If the stack's `nextIndex` counter is greater than or equal to `size`, then evaluating this expression results in a transition to `reject` and sets the error to **`error`**`.StackOutOfBounds`. If `hs` is an l-value, then `hs.next` is also an l-value.
- `hs.last`: produces a reference to the element with index `hs.nextIndex - 1` in the stack, if such an element exists. May only be used in a **`parser`**. If the `nextIndex` counter is less than `1`, or greater than `size`, then evaluating this expression results in a transition to `reject` and sets the error to **`error`**`.StackOutOfBounds`. Unlike `hs.next`, the resulting reference is never an l-value.
- `hs.lastIndex`: produces a 32-bit unsigned integer that encodes the index `hs.nextIndex - 1`. May only be used in a **`parser`**. If the `nextIndex` counter is `0`, then evaluating this expression produces an undefined value.

Finally, P4 offers the following computations that can be used to manipulate the elements at the front and back of the stack:

- `hs.push_front(int count)`: shifts `hs` "right" by `count`. The first `count` elements become invalid. The last `count` elements in the stack are discarded. The `hs.nextIndex` counter is incremented by `count`. The `count` argument must be a compile-time known value that is a positive integer. The return type is **`void`**.
- `hs.pop_front(int count)`: shifts `hs` "left" by `count` (i.e., element with index `count` is copied in stack at index `0`). The last `count` elements become invalid. The `hs.nextIndex` counter is decremented by `count`. The `count` argument must be a compile-time known value that is a positive integer. The return type is **`void`**.

The following pseudocode defines the behavior of `push_front` and `pop_front`:

```
void push_front(int count) {
    for (int i = this.size-1; i >= 0; i -= 1) {
        if (i >= count) {
            this[i] = this[i-count];
        } else {
            this[i].setInvalid();
        }
    }
    this.nextIndex = this.nextIndex + count;
    if (this.nextIndex > this.size) this.nextIndex = this.size;
    // Note: this.last, this.next, and this.lastIndex adjust with this.nextIndex
```

```
}

void pop_front(int count) {
    for (int i = 0; i < this.size; i++) {
        if (i+count < this.size) {
            this[i] = this[i+count];
        } else {
            this[i].setInvalid();
        }
    }
    if (this.nextIndex >= count) {
        this.nextIndex = this.nextIndex - count;
    } else {
        this.nextIndex = 0;
    }
    // Note: this.last, this.next, and this.lastIndex adjust with this.nextIndex
}
```

Similar to structs and headers, the size of a header stack is a compile-time known value (Section 9).

Two header stacks can be compared for equality (`==`) or inequality (`!=`) only if they have the same element type and the same length. Two stacks are equal if and only if all their corresponding elements are equal. Note that the `nextIndex` value is not used in the equality comparison.

8.18.1. Header stack expressions

One can write expressions that evaluate to a header stack. The syntax of these expressions is given by:

```
expression ...
    | '{' expressionList '}'
    | '(' typeRef ')' expression
    ;
```

The `typeRef` is a header stack type. The `typeRef` can be omitted if it can be inferred from context, e.g., when initializing a variable with a header stack type. Each expression in the list must evaluate to a header of the same type as the other stack elements.

Here is an example:

```
header H<T> {
    bit<32> b;
    T t;
}
H<bit<32>>[3] s = (H<bit<32>>[3]){ {0, 1}, {2, 3}, (H<bit<32>>){#} };
// without an explicit cast
H<bit<32>>[3] s1 = { {0, 1}, {2, 3}, (H<bit<32>>){#} };
// using the default initializer
H<bit<32>>[3] s2 = { {0, 1}, {2, 3}, ... };
```

The values of s, s1, and s2 in the above example are identical.

8.19. Operations on header unions

A variable declared with a union type is initially invalid. For example:

```
header H1 {
  bit<8> f;
}
header H2 {
  bit<16> g;
}
header_union U {
  H1 h1;
  H2 h2;
}

U u; // u invalid
```

This also implies that each of the headers h1 through hn contained in a header union are also initially invalid. Unlike headers, a union cannot be initialized. However, the validity of a header union can be updated by assigning a valid header to one of its elements:

```
U u;
H1 my_h1 = { 8w0 }; // my_h1 is valid
u.h1 = my_h1;       // u and u.h1 are both valid
```

We can also assign a tuple to an element of a header union,

```
U u;
u.h2 = { 16w1 };     // u and u.h2 are both valid
```

or set their validity bits directly.

```
U u;
u.h1.setValid();     // u and u.h1 are both valid
H1 my_h1 = u.h1;     // my_h1 is now valid, but contains an undefined value
```

Note that reading an uninitialized header produces an undefined value, even if the header is itself valid.

If u is an expression whose type is a header union U with fields ranged over by hi, then the expression u.hi evaluates to a header, and thus it can be used wherever a header expression is allowed. If u is a left-value, then u.hi is also a left-value.

The following operations:

- u.hi.setValid(): sets the valid bit for header hi to **true** and sets the valid bit for all other headers to **false**, which implies that it is unspecified what value reading any member header of u will return.

- `u.hi.setInvalid()`: if the valid bit for any member header of `u` is **`true`** then sets it to **`false`**, which implies that it is unspecified what value reading any member header of `u` will return.

The assignment to a union field:

```
u.hi = e;
```

has the following meaning:

- if `e` is valid, then it is equivalent to:

```
  u.hi.setValid();
  u.hi = e;
```

- if `e` is invalid, then it is equivalent to:

```
  u.hi.setInvalid();
```

Assignments between variables of the same type of header union are permitted. The assignment `u1 = u2` copies the full state of header union `u2` to `u1`. If `u2` is valid, then there is some header `u2.hi` that is valid. The assignment behaves the same as `u1.hi = u2.hi`. If `u2` is not valid, then `u1` becomes invalid (i.e. if any header of `u1` was valid, it becomes invalid).

`u.isValid()` returns true if any member of the header union `u` is valid, otherwise it returns false. `setValid()` and `setInvalid()` methods are not defined for header unions.

Supplying an expression with a union type to `emit` simply emits the single header that is valid, if any.

The following example shows how we can use header unions to represent IPv4 and IPv6 headers uniformly:

```
header_union IP {
    IPv4 ipv4;
    IPv6 ipv6;
}

struct Parsed_packet {
   Ethernet ethernet;
   IP ip;
}

parser top(packet_in b, out Parsed_packet p) {
    state start {
       b.extract(p.ethernet);
       transition select(p.ethernet.etherType) {
           16w0x0800 : parse_ipv4;
           16w0x86DD : parse_ipv6;
       }
```

```
    }
    state parse_ipv4 {
        b.extract(p.ip.ipv4);
        transition accept;
    }
    state parse_ipv6 {
        b.extract(p.ip.ipv6);
        transition accept;
    }
}
```

As another example, we can also use unions to parse (selected) TCP options:

```
header Tcp_option_end_h {
    bit<8> kind;
}
header Tcp_option_nop_h {
    bit<8> kind;
}
header Tcp_option_ss_h {
    bit<8>  kind;
    bit<32> maxSegmentSize;
}
header Tcp_option_s_h {
    bit<8>  kind;
    bit<24> scale;
}
header Tcp_option_sack_h {
    bit<8>         kind;
    bit<8>         length;
    varbit<256>    sack;
}
header_union Tcp_option_h {
    Tcp_option_end_h  end;
    Tcp_option_nop_h  nop;
    Tcp_option_ss_h   ss;
    Tcp_option_s_h    s;
    Tcp_option_sack_h sack;
}

typedef Tcp_option_h[10] Tcp_option_stack;

struct Tcp_option_sack_top {
    bit<8> kind;
    bit<8> length;
}
```

```
parser Tcp_option_parser(packet_in b, out Tcp_option_stack vec) {
    state start {
        transition select(b.lookahead<bit<8>>()) {
            8w0x0 : parse_tcp_option_end;
            8w0x1 : parse_tcp_option_nop;
            8w0x2 : parse_tcp_option_ss;
            8w0x3 : parse_tcp_option_s;
            8w0x5 : parse_tcp_option_sack;
        }
    }
    state parse_tcp_option_end {
        b.extract(vec.next.end);
        transition accept;
    }
    state parse_tcp_option_nop {
         b.extract(vec.next.nop);
         transition start;
    }
    state parse_tcp_option_ss {
         b.extract(vec.next.ss);
         transition start;
    }
    state parse_tcp_option_s {
         b.extract(vec.next.s);
         transition start;
    }
    state parse_tcp_option_sack {
         bit<8> n = b.lookahead<Tcp_option_sack_top>().length;
         // n is the total length of the TCP SACK option in bytes.
         // The length of the varbit field 'sack' of the
         // Tcp_option_sack_h header is thus n-2 bytes.
         b.extract(vec.next.sack, (bit<32>) (8 * n - 16));
         transition start;
    }
}
```

Similar to headers, the size of a header union is a local compile-time known value (Section 9).

The expression `{#}` represents an invalid header union of some type, but it can be any header or header union type. A P4 compiler may require an explicit cast on this expression in cases where it cannot determine the particular header or header union type from the context.

```
header_union HU { ... }
HU h = (HU){#};  // invalid header union; same as an uninitialized header union.
```

Two header unions can be compared for equality (`==`) or inequality (`!=`) if they have the same type. The unions are equal if and only if all their corresponding fields are equal (i.e., either all fields are invalid

in both unions, or in both unions the same field is valid, and the values of the valid fields are equal as headers).

8.20. Method invocations and function calls

Method invocations and function calls can be invoked using the following syntax:

```
expression
    : ...
    | expression '<' realTypeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList "," argument
    ;

argument
    : expression  /* positional argument */
    | name "=" expression  /* named argument */
    | "_"
    | name "=" "_"
    ;

realTypeArgumentList
    : realTypeArg
    | realTypeArgumentList "," typeArg
    ;

realTypeArg
    : typeRef
    | VOID
    | "_"
    ;
```

A function call or method invocation can optionally specify for each argument the corresponding parameter name. It is illegal to use names only for some arguments: either all or no arguments must specify the parameter name. Function arguments are evaluated in the order they appear, left to right, before the function invocation takes place.

```
extern void f(in bit<32> x, out bit<16> y);
bit<32> xa = 0;
bit<16> ya;
f(xa, ya);  // match arguments by position
f(x = xa, y = ya);  // match arguments by name
f(y = ya, x = xa);  // match arguments by name in any order
//f(x = xa);  -- error: enough arguments
//f(x = xa, x = ya);  -- error: argument specified twice
//f(x = xa, ya);  -- error: some arguments specified by name
//f(z = xa, w = yz);  -- error: no parameter named z or w
//f(x = xa, y = 0);  -- error: y must be a left-value
```

The calling convention is copy-in/copy-out (Section 6.8). For generic functions the type arguments can be explicitly specified in the function call. The compiler only inserts implicit casts for direction **in** arguments to methods or functions as described in Section 8.11. The types for all other arguments must match the parameter types exactly.

The result returned by a function call is discarded when the function call is used as a statement.

The "don't care" identifier (_) can only be used for an **out** function/method argument, when the value of returned in that argument is ignored by subsequent computations. When used in generic functions or methods, the compiler may reject the program if it is unable to infer a type for the don't care argument.

8.21. Constructor invocations

Several P4 constructs denote resources that are allocated at compilation time:

- **extern** objects
- **parser**s
- **control** blocks
- **package**s

Allocation of such objects can be performed in two ways:

- Using constructor invocations, which are expressions that return an object of the corresponding type.
- Using instantiations, described in Section 11.3.

The syntax for a constructor invocation is similar to a function call; constructors can also be called using named arguments. Constructors are evaluated entirely at compilation time (see Section 18). In consequence, all constructor arguments must also be expressions that can be evaluated at compilation time. When performing type inference and overload resolution, constructor invocations are treated similar to methods or functions.

The following example shows a constructor invocation for setting the target-dependent implementation property of a table:

```
extern ActionProfile {
    ActionProfile(bit<32> size);  // constructor
```

```
}
table tbl {
    actions = { /* body omitted */ }
    implementation = ActionProfile(1024);  // constructor invocation
}
```

8.22. Operations on `extern` objects

The only operations that can be performed on **`extern`** objects are the following:

- Instantiating an extern object using a constructor invocation, as described in Section 8.21.
- Invoking a method of an extern object instance using a method call expression, as described in Section 8.20.

Controls, parsers, packages, and extern constructors can have parameters of type **`extern`** only if they are directionless parameters. Extern object instances can thus be used as arguments in the construction of such objects.

No other operations are available on extern objects, e.g., equality comparison is not defined.

8.23. Operations on types introduced by `type`

Values with a type introduced by the **`type`** keyword provide only a few operations:

- assignment to left-values of the same type
- comparisons for equality and inequality if the original type supported such comparisons
- casts to and from the original type

```
type bit<32> U32;
U32 x = (U32)0;  // cast needed
U32 y = (U32) ((bit<32>)x + 1);  // casts needed for arithmetic
bit<32> z = 1;
bool b0 = x == (U32)z; // cast needed
bool b1 = (bit<32>)x == z;  // cast needed
bool b2 = x == y;  // no cast needed
```

8.24. Operations on types that are type variables

Because functions, methods, control, and parsers can be generic, they offer the possibility of declaring values with types that are type variables:

```
control C<T>() {
    apply {
        T x;  // the type of x is T, a type variable
    }
}
```

The type of such objects is not known until the control is instantiated with specific type arguments.

Currently the only operations that are available for such values are assignment (explicit through =, or implicit, through argument passing). This behavior is similar to languages such as Java, and different from languages such as C++.

A future version of P4 may introduce a notion of type constraints which would enable more operations on such values. Because of this limitation, such values are currently of limited utility.

8.25. Reading uninitialized values and writing fields of invalid headers

As mentioned in Section 8.18, any reference to an element of a header stack `hs[index]` where `index` is a compile-time known value must give an error at compile time if the value of the index is out of range. That section also defines the run time behavior of the expressions `hs.next` and `hs.last`, and the behaviors specified there take precedence over anything in this section for those expressions.

All mentions of header stack elements in this section only apply for expressions `hs[index]` where `index` is not a compile-time known value. A P4 implementation may elect not to support expressions of the form `hs[index]` where `index` is not a compile-time known value. However, it does support such expressions, the implementation should conform to the behaviors specified in this section.

The result of reading a value in any of the situations below is that some unspecified value will be used for that field.

- reading a field from a header that is currently invalid.
- reading a field from a header that is currently valid, but the field has not been initialized since the header was last made valid.
- reading any other value that has not been initialized, e.g. a field from a **`struct`**, any uninitialized variable inside of an **`action`** or **`control`**, or an **`out`** parameter of a **`control`** or **`action`** you have called, which was not assigned a value during the execution of that **`control`** or **`action`** (this list of examples is not intended to be exhaustive).
- reading a field of a header that is an element of a header stack, where the index is out of range for the header stack.

Calling the `isValid()` method on an element of a header stack, where the index is out of range, returns an undefined boolean value, i.e., it is either **`true`** or **`false`**, but the specification does not require one or the other, nor that a consistent value is returned across multiple such calls. Assigning an out-of-range header stack element to another header variable `h` leads to a state where `h` is undefined in all of its field values, and its validity is also undefined.

Where a header is mentioned, it may be a member of a **`header_union`**, an element in a header stack, or a normal header. This unspecified value could differ from one such read to another.

For an uninitialized field or variable with a type of **`enum`** or **`error`**, the unspecified value that is read might not be equal to any of the values defined for that type. Such an unspecified value should still lead to predictable behavior in cases where any legal value would match, e.g. it should match in any of these situations:

- If used in a **`select`** expression, it should match **`default`** or _ in a key set expression.
- If used as a key with **`match_kind`** `ternary` in a table, it should match a table entry where the field has all bit positions "don't care".
- If used as a key with **`match_kind`** `lpm` in a table, it should match a table entry where the field has a prefix length of 0.

Consider a situation where a **`header_union`** `u1` has member headers `u1.h1` and `u1.h2`, and at a given point

in the program's execution `u1.h1` is valid and `u1.h2` is invalid. If a write is attempted to a field of the invalid member header `u1.h2`, then any or all of the fields of the valid member header `u1.h1` may change as a result. Such a write must not change the validity of any member headers of `u1`, nor any other state that is currently defined in the system, whether it is defined state in header fields or anywhere else.

If any of these kinds of writes are performed:

- a write to a field in a currently invalid header, either a regular header or an element of a header stack with an index that is in range, and that header is not part of a **`header_union`**
- a write to a field in an element of a header stack, where the index is out of range
- a method call of `setValid()` or `setInvalid()` on an element of a header stack, where the index is out of range

then that write must not change any state that is currently defined in the system, neither in header fields nor anywhere else. In particular, if an invalid header is involved in the write, it must remain invalid.

Any writes to fields in a currently invalid header, or to header stack elements where the index is out of range, are allowed to modify state whose values are not defined, e.g. the values of fields in headers that are currently invalid.

For a top level **`parser`** or **`control`** in an architecture, it is up to that architecture to specify whether parameters with direction **`in`** or **`inout`** are initialized when the control is called, and under what conditions they are initialized, and if so, what their values will be.

Since P4 allows empty tuples and structs, one can construct types whose values carry no "useful" information, e.g.:

```
struct Empty {
   tuple<> t;
}
```

We call the following "empty" types:

- bitstrings with 0 width
- varbits with 0 width
- empty tuples (**`tuple`**<>)
- stacks with 0 size
- structs with no fields
- a tuple having all fields of an empty type
- a struct having all fields of an empty type

Values with empty types carry no useful information. In particular, they do not have to be explicitly initialized to have a legal value.

(Header types with no fields always have a validity bit.)

8.26. Initializing with default values

A left-value can be initialized automatically with a default value of the suitable type using the syntax ... (see Section 7.3). A value of type **`struct`**, **`header`**, or **`tuple`** can also be initialized using a mix of explicit values and default values by using the notation ... in a tuple expression initializer; in this case all fields not explicitly initialized are initialized with default values. When initializing a **`struct`**, **`header`**, and **`tuple`**

with a value containing partially default values using the ... notation the three dots must appear last in the initializer.

```
struct S {
     bit<32> b32;
     bool b;
}

enum int<8> N0 {
   one = 1,
   zero = 0,
   two = 2
}

enum N1 {
     A, B, C, F
}

struct T {
    S s;
    N0 n0;
    N1 n1;
}

header H {
    bit<16> f1;
    bit<8> f2;
}

N0 n0 = ...;  // initialize n0 with the default value 0
N1 n1 = ...;  // initialize n1 with the default value N1.A
S s0 = ...;  // initialize s0 with the default value { 0, false }
S s1 = { 1, ... };  // initialize s1 with the value { 1, false }
S s2 = { b = true, ... }; // initialize s2 with the value { 0, true }
T t0 = ...;  // initialize t0 with the value { { 0, false }, 0, N1.A }
T t1 = { s = ..., ... }; // initialize t1 with the value { { 0, false }, 0, N1.A }
T t2 = { s = ... }; // error: no initializer specified for fields n0 and n1
tuple<N0, N1> p = { ... }; // initialize p with default value { 0, N1.A }
T t3 = { ..., n0 = 2}; // error: ... must be last
H h1 = ...;   // initialize h1 with a header that is invalid
H h2 = { f2=5, ... };   // initialize h2 with a header that is valid, field f1 0, field f2 5
H h3 = { ... };  // initialize h3 with a header that is valid, field f1 0, field f2 0
```

9. Compile-time size determination

The method calls `minSizeInBits`, `minSizeInBytes`, `maxSizeInBits`, and `maxSizeInBytes` can be applied to certain expressions. These method calls return the minimum (or maximum) size in bits (or bytes) required to store the expression. Thus, the result type of these methods has type **`int`**. Except in certain situations involving type variables, discussed below, these method calls produce local compile-time known values; otherwise they produce compile-time known values. None of these methods evaluate the expression that is the receiver of the method call, so it may be invalid (e.g., an out-of-bounds header stack access).

The method `minSizeInBytes` returns the result of `minSizeInBits` rounded up to the next whole number of bytes. In other words, for any expression `e`, `e.minSizeInBytes()` is equal to `(e.minSizeInBits() + 7) >> 3`.

The method `maxSizeInBytes` always returns the result of `maxSizeInBits` rounded up to the next whole number of bytes. In other words, for any expression `e`, `e.maxSizeInBytes()` is equal to `(e.maxSizeInBits() + 7) >> 3`.

The definition of `e.minSizeInBits()` and `e.maxSizeInBits()` is given recursively on the type of `e` as described in the following table:

Type	minSizeInBits	maxSizeInBits
`bit`<N>	N	N
`int`<N>	N	N
`bool`	1	1
`enum bit`<N>	N	N
`enum int`<N>	N	N
`tuple`	foreach field(tuple) sum of `field.minSizeInBits()`	foreach field(tuple) sum of `field.maxSizeInBits()`
`varbit`<N>	0	N
`struct`	foreach field(struct) sum of `field.minSizeInBits()`	foreach field(struct) sum of `field.maxSizeInBits()`
`header`	foreach field(header) sum of `field.minSizeInBits()`	foreach field(header) sum of `field.maxSizeInBits()`
`H[N]`	`N * H.minSizeInBits()`	`N * H.maxSizeInBits()`
`header_union`	max(foreach field(header_union) `field.minSizeInBits()`)	max(foreach field(header_union) `field.maxSizeInBits()`)

The methods can also be applied to type name expressions `e`:

- if the type of `e` is a type introduced by **`type`**, the result is the application of the method to the underlying type
- if `e` is the name of a type (e.g., introduced by a **`typedef`** declaration), where the type given a name is one of the above, then the result is obtained by applying the method to the underlying type.

These methods are defined for:

- all serializable types
- for a type that does not contain **`varbit`** fields, both methods return the same result
- for a type that does contain **`varbit`** fields, `maxSizeInBits` is the worst-case size of the serialized representation of the data and `minSizeInBits` is the "best" case.

Every other case is undefined and will produce a compile-time error. In particular, cases involving type

variables produce a compile-time error.

10. Function declarations

Functions can only be declared at the top level and all parameters must have a direction. P4 functions are modeled after functions as found in most other programming languages, but the language does not permit recursive functions.

```
functionDeclaration
    : annotations functionPrototype blockStatement
    | functionPrototype blockStatement
    ;

functionPrototype
    : typeOrVoid name optTypeParameters "(" parameterList ")"
    ;
```

Here is an example of a function that returns the maximum of two 32-bit values:

```
bit<32> max(in bit<32> left, in bit<32> right) {
   return (left > right) ? left : right;
}
```

A function returns a value using the **return** statement. A function with a return type of **void** can simply use the **return** statement with no arguments. A function with a non-void return type must return a value of the suitable type on all possible execution paths.

11. Constants and variable declarations

11.1. Constants

Constant values are defined with the syntax:

```
constantDeclaration
    : optAnnotations CONST typeRef name "=" initializer ";"
    ;

initializer
    : expression
    ;
```

Such a declaration introduces a constant whose value has the specified type. The following are all legal constant declarations:

```
const bit<32> COUNTER = 32w0x0;
struct Version {
```

```
    bit<32> major;
    bit<32> minor;
}
const Version version = { 32w0, 32w0 };
```

The `initializer` expression must be a compile-time known value.

11.2. Variables

Local variables are declared with a type, a name, and an optional initializer (as well as an optional annotation):

```
variableDeclaration
    : annotations typeRef name optInitializer ";"
    | typeRef name optInitializer ";"
    ;

optInitializer
    : /* empty */
    | "=" initializer
    ;
```

Variable declarations without an initializer are uninitialized (except for headers and other header-related types, which are initialized to invalid in the same way as described for direction **out** parameters in Section 6.8). The language places few restrictions on the types of the variables: most P4 types that can be written explicitly can be used (e.g., base types, **struct**, **header**, header stack, **tuple**). However, it is impossible to declare variables with type **int**, or with types that are only synthesized by the compiler (e.g., `set`) In addition, variables of type **parser**, **control**, **package**, or **extern** types must be declared using instantiations (see Section 11.3).

Reading the value of a variable that has not been initialized yields an undefined result. The compiler should attempt to detect and emit a warning in such situations.

Variables declarations can appear in the following locations within a P4 program:

- In a block statement,
- In a **parser** state,
- In an **action** body,
- In a **control** block's **apply** sub-block,
- In the list of local declarations in a **parser**, and
- In the list of local declarations in a **control**.

Variables have local scope, and behave like stack-allocated variables in languages such as C. The value of a variable is never preserved from one invocation of its enclosing block to the next. In particular, variables cannot be used to maintain state between different network packets.

11.3. Instantiations

Instantiations are similar to variable declarations, but are reserved for the types with constructors (**extern** objects, **control** blocks, **parser**s, and **package**s):

```
instantiation
    : typeRef '(' argumentList ')' name ';'
    | annotations typeRef '(' argumentList ')' name ';'
    ;
```

An instantiation is written as a constructor invocation followed by a name. Instantiations are always executed at compilation time (Section 18.1). The effect is to allocate an object with the specified name, and to bind it to the result of the constructor invocation. Note that instantiation arguments can be specified by name.

For example, a hypothetical bank of counter objects can be instantiated as follows:

```
// from target library
enum CounterType {
   Packets,
   Bytes,
   Both
}
extern Counter {
    Counter(bit<32> size, CounterType type);
    void increment(in bit<32> index);
}
// user program
control c(/* parameters omitted */) {
    Counter(32w1024, CounterType.Both) ctr;  // instantiation
    apply { /* body omitted */ }
}
```

11.3.1. Instantiating objects with abstract methods

When instantiating an extern type that has **abstract** methods users have to supply implementations for all such methods. This is done using object initializers:

```
lvalue:
    ...
    | THIS

expression:
    ...
    | THIS

instantiation:
      ...
      | annotations typeRef "(" argumentList ")" name "=" objInitializer ";"
      | typeRef "(" argumentList ")" name "=" objInitializer ";"
```

```
objInitializer
    : "{" objDeclarations "}"
    ;

objDeclarations
    : /* empty */
    | objDeclarations objDeclaration
    ;

objDeclaration
    : functionDeclaration
    | instantiation
    ;
```

The abstract methods can only use the supplied arguments or refer to values that are in the top-level scope. When calling another method of the same instance the **this** keyword is used to indicate the current object instance:

```
// Instantiate a balancer
Balancer() b = {  // provide an implementation for the abstract methods
    bit<4> on_new_flow(in bit<32> address) {
        // uses the address and the number of flows to balance the load
        bit<32> count = this.getFlowCount();  // call method of the same instance
        return (address + count)[3:0];
    }
}
```

Abstract methods may be invoked by users explicitly, or they may be invoked by the target architecture. The architectural description has to specify when the abstract methods are invoked and what the meaning of their arguments and return values is; target architectures may impose additional constraints on abstract methods.

11.3.2. Restrictions on top-level instantiations

A P4 program may not instantiate controls and parsers in the top-level package. This restriction is designed to ensure that most state resides in the architecture itself, or is local to a **parser** or **control**. For example, the following program is not valid:

```
// Program
control c(/* parameters omitted */) { /* body omitted */ }
c() c1;  // illegal top-level instantiation
```

because control `c1` is instantiated at the top-level. Note that top-level declarations of constants and instantiations of extern objects are permitted.

12. Statements

Every statement in P4 except block statements must end with a semicolon. Statements can appear in several places:

- Within **parser** states
- Within a **control** block
- Within an **action**

There are restrictions for the kinds of statements that can appear in each of these places. For example, **return**s are not supported in parsers, and **switch** statements are only supported in control blocks. We present here the most general case, for control blocks.

```
statement
    : assignmentOrMethodCallStatement
    | conditionalStatement
    | emptyStatement
    | blockStatement
    | exitStatement
    | returnStatement
    | switchStatement
    ;

assignmentOrMethodCallStatement
    : lvalue "(" argumentList ")" ";"
    | lvalue "<" typeArgumentList ">" "(" argumentList ")" ";"
    | lvalue "="  expression ";"
    ;
```

In addition, parsers support a **transition** statement (Section 13.5).

12.1. Assignment statement

An assignment, written with the = sign, first evaluates its left sub-expression to an l-value, then evaluates its right sub-expression to a value, and finally copies the value into the l-value. Derived types (e.g. `structs`) are copied recursively, and all components of **header**s are copied, including "validity" bits. Assignment is not defined for **extern** values.

12.2. Empty statement

The empty statement, written ; is a no-op.

```
emptyStatement
    : ";"
    ;
```

12.3. Block statement

A block statement is denoted by curly braces. It contains a sequence of statements and declarations, which are executed sequentially. The declarations (e.g., variables and constants) within a block statement are only visible within the block.

```
blockStatement
    : optAnnotations "{" statOrDeclList "}"
    ;

statOrDeclList
    : /* empty */
    | statOrDeclList statementOrDeclaration
    ;

statementOrDeclaration
    : variableDeclaration
    | constantDeclaration
    | statement
    ;
```

12.4. Return statement

The **return** statement immediately terminates the execution of the **action**, function or **control** containing it. **return** statements are not allowed within parsers. **return** statements followed by an expression are only allowed within functions that return values; in this case the type of the expression must match the return type of the function. Any copy-out behavior due to direction **out** or **inout** parameters of the enclosing **action**, function, or **control** are still performed after the execution of the **return** statement. See Section 6.8 for details on copy-out behavior.

```
returnStatement
    : RETURN ";"
    | RETURN expression ";"
    ;
```

12.5. Exit statement

The **exit** statement immediately terminates the execution of all the blocks currently executing: the current **action** (if invoked within an **action**), the current **control**, and all its callers. **exit** statements are not allowed within parsers or functions.

Any copy-out behavior due to direction **out** or **inout** parameters of the enclosing **action** or **control**, and all of its callers, are still performed after the execution of the **exit** statement. See Section 6.8 for details on copy-out behavior.

```
exitStatement
    : EXIT ";"
    ;
```

There are some expressions whose evaluation might cause an **exit** statement to be executed. Examples include:

- **table.apply**().action_run, which can only appear as the expression of a **switch** statement (see Section 12.7), and when it appears there, it must be the entire expression.
- Any expression containing **table.apply**().hit or **table.apply**().miss (see Section 14.2.2), which can be part of arbitrarily complex expressions in many places of a P4 program, such as:
  - the expression in an **if** statement.
  - the expression e1 in a conditional expression e1 ? e2 : e3.
  - in an assignment statement, in the left and/or right hand sides.
  - an argument passed to a function or method call.
  - an expression to calculate a table key (see Section 14.2.3).

This list is not intended to be exhaustive.

If applying the table causes an action to be executed, which in turn causes an **exit** statement to be executed, then evaluation of the expression ends immediately, and the rest of the current expression or statement does not complete its execution. See Section 8.1 for the order of evaluation of the parts of an expression. For the examples listed above, it also means the following behavior after the expression evaluation is interrupted.

- For a **switch** statement, if **table.apply**() exits, then none of the blocks in the **switch** statement are executed.
- If **table.apply**().hit or **table.apply**().miss cause an exit during the evaluation of an expression:
  - If it is the expression of an **if** statement, then neither the 'then' nor 'else' branches of the **if** statement are executed.
  - If it is the expression e1 in a conditional expression e1 ? e2 : e3, then neither expression e2 nor e3 are evaluated.
  - If the expression is the right hand side of an assignment statement, or part of the calculation of the L-value on the left hand side (e.g. the index expression of a header stack reference), then no assignment occurs.
  - If the expression is an argument passed to a function or method call, then the function/method call does not occur.
  - If the expression is a table key, then the table is not applied.

12.6. Conditional statement

The conditional statement uses standard syntax and semantics familiar from many programming languages.

However, the condition expression in P4 is required to be a Boolean (and not an integer).

```
conditionalStatement
    : IF "(" expression ")" statement
```

```
    | IF "(" expression ")" statement ELSE statement
    ;
```

When several **if** statements are nested, the **else** applies to the innermost **if** statement that does not have an **else** statement.

12.7. Switch statement

The **switch** statement can only be used within **control** blocks.

```
switchStatement
    : SWITCH "(" expression ")" "{" switchCases "}"
    ;

switchCases
    : /* empty */
    | switchCases switchCase
    ;

switchCase
    : switchLabel ":" blockStatement
    | switchLabel ":"  // fall-through
    ;

switchLabel
    : DEFAULT
    | nonBraceExpression
    ;

nonBraceExpression
    : INTEGER
    | STRING_LITERAL
    | TRUE
    | FALSE
    | THIS
    | prefixedNonTypeName
    | nonBraceExpression "[" expression "]"
    | nonBraceExpression "[" expression ":" expression "]"
    | "(" expression ")"
    | "!" expression %prec PREFIX
    | "~" expression %prec PREFIX
    | "-" expression %prec PREFIX
    | "+" expression %prec PREFIX
    | typeName "." member
    | ERROR "." member
    | nonBraceExpression "." member
```

```
    | nonBraceExpression "*" expression
    | nonBraceExpression "/" expression
    | nonBraceExpression "%" expression
    | nonBraceExpression "+" expression
    | nonBraceExpression "-" expression
    | nonBraceExpression "|+|" expression
    | nonBraceExpression "|-|" expression
    | nonBraceExpression "<<" expression
    | nonBraceExpression ">>" expression
    | nonBraceExpression "<=" expression
    | nonBraceExpression ">=" expression
    | nonBraceExpression "<" expression
    | nonBraceExpression ">" expression
    | nonBraceExpression "!=" expression
    | nonBraceExpression "==" expression
    | nonBraceExpression "&" expression
    | nonBraceExpression "^" expression
    | nonBraceExpression "|" expression
    | nonBraceExpression "++" expression
    | nonBraceExpression "&&" expression
    | nonBraceExpression "||" expression
    | nonBraceExpression "?" expression ":" expression
    | nonBraceExpression "<" realTypeArgumentList ">" "(" argumentList ")"
    | nonBraceExpression "(" argumentList ")"
    | namedType "(" argumentList ")"
    | "(" typeRef ")" expression
    ;
```

The `nonBraceExpression` is the same as `expression` as defined in Section 8, except it does not include any cases that can begin with a left brace `{` character, to avoid syntactic ambiguity with a block statement.

There are two kinds of **`switch`** expressions allowed, described separately in the following two subsections.

12.7.1. Switch statement with `action_run` expression

For this variant of **`switch`** statement, the expression must be of the form `t.`**`apply`**`().action_run`, where `t` is the name of a table (see Section 14.2.2). All switch labels must be names of actions of the table `t`, or **`default`**.

```
switch (t.apply().action_run) {
   action1:          // fall-through to action2:
   action2: { /* body omitted */ }
   action3: { /* body omitted */ }  // no fall-through from action2 to action3 labels
   default: { /* body omitted */ }
}
```

Note that the **`default`** label of the **`switch`** statement is used to match on the kind of action executed, no

matter whether there was a table hit or miss. The **default** label does not indicate that the table missed and the `default_action` was executed.

12.7.2. Switch statement with integer or enumerated type expression

For this variant of **switch** statement, the expression must evaluate to a result with one of these types:

- **bit**<W>
- **int**<W>
- **enum**, either with or without an underlying representation specified
- **error**

All switch labels must be expressions with compile-time known values, and must have a type that can be implicitly cast to the type of the **switch** expression (see Section 8.11.2). Switch labels must not begin with a left brace character `{`, to avoid ambiguity with a block statement.

```
// Assume the expression hdr.ethernet.etherType has type bit<16>
switch (hdr.ethernet.etherType) {
   0x86dd: { /* body omitted */ }
   0x0800:           // fall-through to the next body
   0x0802: { /* body omitted */ }
   0xcafe: { /* body omitted */ }
   default: { /* body omitted */ }
}
```

12.7.3. Notes common to all switch statements

It is a compile-time error if two labels of a **switch** statement equal each other. The switch label values need not include all possible values of the switch expression. It is optional to have a **switch** case with the **default** label, but if one is present, it must be the last one in the **switch** statement.

If a switch label is not followed by a block statement, it falls through to the next label. However, if a block statement is present, it does not fall through. Note that this is different from C-style **switch** statements, where a `break` is needed to prevent fall-through. If the last switch label is not followed by a block statement, the behavior is the same as if the last switch label were followed by an empty block statement `{ }`.

When a **switch** statement is executed, first the switch expression is evaluated, and any side effects from evaluating this expression are visible to any **switch** case that is executed. Among switch labels that are not **default**, at most one of them can equal the value of the switch expression. If one is equal, that switch case is executed.

If no labels are equal to the **switch** expression, then:

- if there is a **default** label, the case with the **default** label is executed.
- if there is no **default** label, then no switch case is executed, and execution continues after the end of the **switch** statement, with no side effects (except any that were caused by evaluating the **switch** expression).

See "Implementing generalized P4_16 switch statements" GeneralizedSwitchStatements for possible techniques that one might use to implement generalized switch statements.

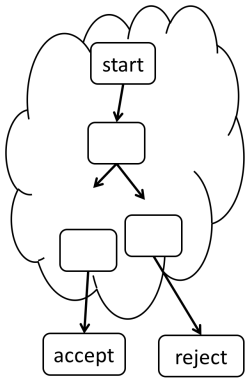

Figure 8. Parser FSM structure.

13. Packet parsing

This section describes the P4 constructs specific to parsing network packets.

13.1. Parser states

A P4 parser describes a state machine with one start state and two final states. The start state is always named `start`. The two final states are named `accept` (indicating successful parsing) and `reject` (indicating a parsing failure). The `start` state is part of the parser, while the `accept` and `reject` states are distinct from the states provided by the programmer and are logically outside of the parser. Figure 8 illustrates the general structure of a parser state machine.

13.2. Parser declarations

A parser declaration comprises a name, a list of parameters, an optional list of constructor parameters, local elements, and parser states (as well as optional annotations).

```
parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters
      "(" parameterList ")"
    ;

parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      "{" parserLocalElements parserStates "}"
    ;

parserLocalElements
    : /* empty */
    | parserLocalElements parserLocalElement
    ;
```

```
parserStates
    : parserState
    | parserStates parserState
    ;
```

For a description of `optConstructorParameters`, which are useful for building parameterized parsers, see Section 15.

Unlike parser type declarations, parser declarations may not be generic—e.g., the following declaration is illegal:

```
parser P<H>(inout H data) { /* body omitted */ }
```

Hence, used in the context of a `parserDeclaration` the production rule `parserTypeDeclaration` should not yield type parameters.

At least one state, named `start`, must be present in any **`parser`**. A parser may not define two states with the same name. It is also illegal for a parser to give explicit definitions for the `accept` and `reject` states—those states are logically distinct from the states defined by the programmer.

State declarations are described below. Preceding the parser states, a **`parser`** may also contain a list of local elements. These can be constants, variables, or instantiations of objects that may be used within the parser. Such objects may be instantiations of **`extern`** objects, or other **`parser`**s that may be invoked as subroutines. However, it is illegal to instantiate a **`control`** block within a **`parser`**.

```
parserLocalElement
    : constantDeclaration
    | instantiation
    | variableDeclaration
    | valueSetDeclaration
    ;
```

The states and local elements are all in the same namespace, thus the following example will produce an error:

```
// erroneous example
parser p() {
    bit<4> t;
    state start {
       t = 1;
       transition t;
    }
    state t {  // error: name t is duplicated
       transition accept;
    }
}
```

For an example containing a complete declaration of a parser see Section 5.3.

13.3. The Parser abstract machine

The semantics of a P4 parser can be formulated in terms of an abstract machine that manipulates a `ParserModel` data structure. This section describes this abstract machine in pseudo-code.

A parser starts execution in the `start` state and ends execution when one of the `reject` or `accept` states has been reached.

```
ParserModel {
    error       parseError;
    onPacketArrival(packet p) {
        ParserModel.parseError = error.NoError;
        goto start;
    }
}
```

An architecture must specify the behavior when the `accept` and `reject` states are reached. For example, an architecture may specify that all packets reaching the `reject` state are dropped without further processing. Alternatively, it may specify that such packets are passed to the next block after the parser, with intrinsic metadata indicating that the parser reached the `reject` state, along with the error recorded.

13.4. Parser states

A parser state is declared with the following syntax:

```
parserState
    : optAnnotations STATE name
      "{" parserStatements transitionStatement "}"
    ;
```

Each state has a name and a body. The body consists of a sequence of statements that describe the processing performed when the parser transitions to that state, including:

- Local variable declarations,
- Assignment statements,
- Method calls, which serve several purposes:
  - Invoking functions (e.g., using **`verify`** to check the validity of data already parsed), and
  - Invoking methods (e.g., extracting data out of packets or computing checksums) and other parsers (see Section 13.10), and
- Conditional statements,
- Transitions to other states (discussed in Section 13.5).

The syntax for parser statements is given by the following grammar rules:

```
parserStatements
    : /* empty */
    | parserStatements parserStatement
    ;
```

```
parserStatement
    : assignmentOrMethodCallStatement
    | directApplication
    | emptyStatement
    | variableDeclaration
    | constantDeclaration
    | parserBlockStatement
    | conditionalStatement
    ;

parserBlockStatement
    : optAnnotations "{" parserStatements "}"
    ;
```

Architectures may place restrictions on the expressions and statements that can be used in a parser—e.g., they may forbid the use of operations such as multiplication or place restrictions on the number of local variables that may be used.

In terms of the `ParserModel`, the sequence of statements in a state are executed sequentially.

13.5. Transition statements

The last statement in a parser state is an optional **`transition`** statement, which transfers control to another state, possibly `accept` or `reject`. A **`transition`** statements is written using the following syntax:

```
transitionStatement
    : /* empty */
    | TRANSITION stateExpression
    ;

stateExpression
    : name ";"
    | selectExpression
    ;
```

The execution of the transition statement causes `stateExpression` to be evaluated, and transfers control to the resulting state.

In terms of the `ParserModel`, the semantics of a **`transition`** statement can be formalized as follows:

```
goto eval(stateExpression)
```

For example, this statement:

```
transition accept;
```

terminates execution of the current parser and transitions immediately to the `accept` state.

If the body of a state block does not end with a **`transition`** statement, the implied statement is

```
transition reject;
```

13.6. Select expressions

A **select** expression evaluates to a state. The syntax for a **select** expression is as follows:

```
selectExpression
    : SELECT "(" expressionList ")" "{" selectCaseList "}"
    ;

selectCaseList
    : /* empty */
    | selectCaseList selectCase
    ;

selectCase
    : keysetExpression ":" name ";"
    ;
```

Each expression in the `expressionList` must have a type of **bit**<W>, **int**<W>, **bool**, **enum**, serializable **enum**, or a **tuple** type with fields of one of the above types.

In a **select** expression, if the `expressionList` has type **tuple**<T>, then each `keysetExpression` must have type `set<`**tuple**`<T>>`. In particular, if a set is specified as a range or mask expression, the endpoints of the range and mask expression are implicitly cast to type `T` using the standard rules for casts.

In terms of the `ParserModel`, the meaning of a select expression:

```
select(e) {
    ks[0]: s[0];
    ks[1]: s[1];
    /* more labels omitted */
    ks[n-2]: s[n-1];
    _ : sd;  // ks[n-1] is default
}
```

is defined in pseudo-code as:

```
key = eval(e);
for (int i=0; i < n; i++) {
    keyset = eval(ks[i]);
    if (keyset.contains(key)) return s[i];
}
verify(false, error.NoMatch);
```

Some targets may require that all keyset expressions in a select expression be compile-time known values. Keysets are evaluated in order, from top to bottom as implied by the pseudo-code above; the

first keyset that includes the value in the **select** argument provides the result state. If no label matches, the execution triggers a runtime error with the standard error code **error**.NoMatch.

Note that this implies that all cases after a **default** or _ label are unreachable; the compiler should emit a warning if it detects unreachable cases. This constitutes an important difference between **select** expressions and the **switch** statements found in many programming languages since the keysets of a **select** expression may "overlap".

The typical way to use a **select** expression is to compare the value of a recently-extracted header field against a set of values, as in the following example:

```
header IPv4_h { bit<8> protocol; /* more fields omitted */ }
struct P { IPv4_h ipv4; /* more fields omitted */ }
P headers;
select (headers.ipv4.protocol) {
    8w6  : parse_tcp;
    8w17 : parse_udp;
    _    : accept;
}
```

For example, to detect TCP reserved ports (< 1024) one could write:

```
select (p.tcp.port) {
    16w0 &&& 16w0xFC00: well_known_port;
    _: other_port;
}
```

The expression `16w0 &&& 16w0xFC00` describes the set of 16-bit values whose most significant six bits are zero.

Some targets may support parser value sets; see Section 13.11. Given a type `T` for the type parameter of the value set, the type of the value set is `set<T>`. The type of the value set must match to the type of all other `keysetExpressions` in the same **select** expression. If there is a mismatch, the compiler must raise an error. The type of the values in the set must be either bit<>, int<>, tuple, struct, or serializable **enum**.

For example, to allow the control plane API to specify TCP reserved ports at runtime, one could write:

```
struct vsk_t {
    @match(ternary)
    bit<16> port;
}
value_set<vsk_t>(4) pvs;
select (p.tcp.port) {
    pvs: runtime_defined_port;
    _: other_port;
}
```

The above example allows the runtime API to populate up to 4 different `keysetExpressions` in the **value_set**. If the **value_set** takes a struct as type parameter, the runtime API can use the struct field names to name the objects in the value set. The match type of the struct field is specified with the `@match` annotation. If

the @match annotation is not specified on a struct field, by default it is assumed to be @match(exact). A single non-exact field must be placed into a struct by itself, with the desired @match annotation.

13.7. verify

The **verify** statement provides a simple form of error handling. **verify** can only be invoked within a parser; it is used syntactically as if it were a function with the following signature:

```
extern void verify(in bool condition, in error err);
```

If the first argument is **true**, then executing the statement has no side-effect. However, if the first argument is **false**, it causes an immediate transition to reject, which causes immediate parsing termination; at the same time, the parserError associated with the parser is set to the value of the second argument.

In terms of the ParserModel the semantics of a **verify** statement is given by:

```
ParserModel.verify(bool condition, error err) {
    if (condition == false) {
        ParserModel.parserError = err;
        goto reject;
    }
}
```

13.8. Data extraction

The P4 core library contains the following declaration of a built-in **extern** type called packet_in that represents incoming network packets. The packet_in extern is special: it cannot be instantiated by the user explicitly. Instead, the architecture supplies a separate instance for each packet_in argument to a **parser** instantiation.

```
extern packet_in {
    void extract<T>(out T headerLvalue);
    void extract<T>(out T variableSizeHeader, in bit<32> varFieldSizeBits);
    T lookahead<T>();
    bit<32> length();  // This method may be unavailable in some architectures
    void advance(bit<32> bits);
}
```

To extract data from a packet represented by an argument b with type packet_in, a parser invokes the extract methods of b. There are two variants of the extract method: a one-argument variant for extracting fixed-size headers, and a two-argument variant for extracting variable-sized headers. Because these operations can cause runtime verification failures (see below), these methods can only be executed within parsers.

When extracting data into a bit-string or integer, the first packet bit is extracted to the most significant bit of the integer.

Some targets may perform cut-through packet processing, i.e., they may start processing a packet before its length is known (i.e., before all bytes have been received). On such a target calls to the

`packet_in.length()` method cannot be implemented. Attempts to call this method should be flagged as errors (either at compilation time by the compiler back-end, or when attempting to load the compiled P4 program onto a target that does not support this method).

In terms of the `ParserModel`, the semantics of `packet_in` can be captured using the following abstract model of packets:

```
packet_in {
    unsigned nextBitIndex;
    byte[] data;
    unsigned lengthInBits;
    void initialize(byte[] data) {
        this.data = data;
        this.nextBitIndex = 0;
        this.lengthInBits = data.sizeInBytes * 8;
    }
    bit<32> length() { return this.lengthInBits / 8; }
}
```

13.8.1. Fixed-width extraction

The single-argument `extract` method handles fixed-width headers, and is declared in P4 as follows:

```
void extract<T>(out T headerLeftValue);
```

The expression `headerLeftValue` must evaluate to an l-value (see Section 6.7) of type **`header`** with a fixed width. If this method executes successfully, on completion the `headerLvalue` is filled with data from the packet and its validity bit is set to **`true`**. This method may fail in various ways—e.g., if there are not enough bits left in the packet to fill the specified header.

For example, the following program fragment extracts an Ethernet header:

```
struct Result { Ethernet_h ethernet;  /* more fields omitted */ }
parser P(packet_in b, out Result r) {
    state start {
        b.extract(r.ethernet);
    }
}
```

In terms of the `ParserModel`, the semantics of the single-argument `extract` is given in terms of the following pseudo-code method, using data from the `packet` class defined above. We use the special `valid$` identifier to indicate the hidden valid bit of a header, `isNext$` to indicate that the l-value was obtained using `next`, and `nextIndex$` to indicate the corresponding header or header union stack properties.

```
void packet_in.extract<T>(out T headerLValue) {
   bitsToExtract = sizeofInBits(headerLValue);
   lastBitNeeded = this.nextBitIndex + bitsToExtract;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
```

```
    headerLValue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
    headerLValue.valid$ = true;
    if headerLValue.isNext$ {
      verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
      headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
    }
    this.nextBitIndex += bitsToExtract;
}
```

13.8.2. Variable-width extraction

The two-argument `extract` handles variable-width headers, and is declared in P4 as follows:

```
void extract<T>(out T headerLvalue, in bit<32> variableFieldSize);
```

The expression `headerLvalue` must be an l-value representing a header that contains exactly one **`varbit`** field. The expression `variableFieldSize` must evaluate to a **`bit`**<32> value that indicates the number of bits to be extracted into the unique **`varbit`** field of the header (i.e., this size is not the size of the complete header, just the **`varbit`** field).

In terms of the `ParserModel`, the semantics of the two-argument `extract` is captured by the following pseudo-code:

```
void packet_in.extract<T>(out T headerLvalue,
                          in bit<32> variableFieldSize) {
    // targets are allowed to include the following line, but need not
    // verify(variableFieldSize[2:0] == 0, error.ParserInvalidArgument);
    bitsToExtract = sizeOfFixedPart(headerLvalue) + variableFieldSize;
    lastBitNeeded = this.nextBitIndex + bitsToExtract;
    ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
    ParserModel.verify(bitsToExtract <= headerLvalue.maxSize, error.HeaderTooShort);
    headerLvalue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
    headerLvalue.varbitField.size = variableFieldSize;
    headerLvalue.valid$ = true;
    if headerLValue.isNext$ {
      verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
      headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
    }
    this.nextBitIndex += bitsToExtract;
}
```

The following example shows one way to parse IPv4 options—by splitting the IPv4 header into two separate headers:

```
// IPv4 header without options
header IPv4_no_options_h {
    bit<4>   version;
```

```
    bit<4>   ihl;
    bit<8>   diffserv;
    bit<16>  totalLen;
    bit<16>  identification;
    bit<3>   flags;
    bit<13>  fragOffset;
    bit<8>   ttl;
    bit<8>   protocol;
    bit<16>  hdrChecksum;
    bit<32>  srcAddr;
    bit<32>  dstAddr;
}
header IPv4_options_h {
    varbit<320> options;
}

struct Parsed_headers {
     // Some fields omitted
     IPv4_no_options_h ipv4;
     IPv4_options_h    ipv4options;
}

error { InvalidIPv4Header }

parser Top(packet_in b, out Parsed_headers headers) {
    // Some states omitted

    state parse_ipv4 {
        b.extract(headers.ipv4);
        verify(headers.ipv4.ihl >= 5, error.InvalidIPv4Header);
        transition select (headers.ipv4.ihl) {
            5: dispatch_on_protocol;
            _: parse_ipv4_options;
        }
    }

    state parse_ipv4_options {
        // use information in the ipv4 header to compute the number of bits to extract
        b.extract(headers.ipv4options,
                  (bit<32>)(((bit<16>)headers.ipv4.ihl - 5) * 32));
        transition dispatch_on_protocol;
    }
}
```

13.8.3. Lookahead

The `lookahead` method provided by the `packet_in` packet abstraction evaluates to a set of bits from the input packet without advancing the `nextBitIndex` pointer. Similar to `extract`, it will transition to `reject` and set the error if there are not enough bits in the packet. When `lookahead` returns a value that contains headers (e.g., a header type, or a struct containing headers), the headers values in the returned result are always valid (otherwise `lookahead` must have transitioned to the `reject` state).

The `lookahead` method can be invoked as follows:

```
b.lookahead<T>()
```

where `T` must be a type with fixed width. In case of success the result of the evaluation of `lookahead` returns a value of type `T`.

In terms of the `ParserModel`, the semantics of `lookahead` is given by the following pseudocode:

```
T packet_in.lookahead<T>() {
   bitsToExtract = sizeof(T);
   lastBitNeeded = this.nextBitIndex + bitsToExtract;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   T tmp = this.data.extractBits(this.nextBitIndex, bitsToExtract);
   return tmp;
}
```

The TCP options example from Section 8.19 also illustrates how `lookahead` can be used:

```
state start {
    transition select(b.lookahead<bit<8>>()) {
        0: parse_tcp_option_end;
        1: parse_tcp_option_nop;
        2: parse_tcp_option_ss;
        3: parse_tcp_option_s;
        5: parse_tcp_option_sack;
    }
}

// Some states omitted

state parse_tcp_option_sack {
    bit<8> n = b.lookahead<Tcp_option_sack_top>().length;
    b.extract(vec.next.sack, (bit<32>) (8 * n - 16));
    transition start;
}
```

13.8.4. Skipping bits

P4 provides two ways to skip over bits in an input packet without assigning them to a header:

One way is to `extract` to the underscore identifier, explicitly specifying the type of the data:

```
b.extract<T>(_)
```

Another way is to use the `advance` method of the packet when the number of bits to skip is known.

In terms of the `ParserModel`, the meaning of `advance` is given in pseudocode as follows:

```
void packet_in.advance(bit<32> bits) {
   // targets are allowed to include the following line, but need not
   // verify(bits[2:0] == 0, error.ParserInvalidArgument);
   lastBitNeeded = this.nextBitIndex + bits;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   this.nextBitIndex += bits;
}
```

13.9. Header stacks

A header stack has two properties, `next` and `last`, which can be used in parsing. Consider the following declaration, which defines a stack for representing the headers of a packet with at most ten MPLS headers:

```
header Mpls_h {
    bit<20> label;
    bit<3>  tc;
    bit     bos;
    bit<8>  ttl;
}
Mpls_h[10] mpls;
```

The expression `mpls.next` represents an l-value of type `Mpls_h` that references an element in the `mpls` stack. Initially, `mpls.next` refers to the first element of stack. It is automatically advanced on each successful call to `extract`. The `mpls.last` property refers to the element immediately preceding `next` if such an element exists. Attempting to access `mpls.next` element when the stack's `nextIndex` counter is greater than or equal to `size` causes a transition to `reject` and sets the error to **`error`**`.StackOutOfBounds`. Likewise, attempting to access `mpls.last` when the `nextIndex` counter is equal to `0` causes a transition to `reject` and sets the error to **`error`**`.StackOutOfBounds`.

The following example shows a simplified parser for MPLS processing:

```
struct Pkthdr {
   Ethernet_h ethernet;
   Mpls_h[3] mpls;
   // other headers omitted
}

parser P(packet_in b, out Pkthdr p) {
    state start {
        b.extract(p.ethernet);
```

```
        transition select(p.ethernet.etherType) {
           0x8847: parse_mpls;
           0x0800: parse_ipv4;
        }
    }
    state parse_mpls {
         b.extract(p.mpls.next);
         transition select(p.mpls.last.bos) {
            0: parse_mpls; // This creates a loop
            1: parse_ipv4;
         }
    }
    // other states omitted
}
```

13.10. Sub-parsers

P4 allows parsers to invoke the services of other parsers, similar to subroutines. To invoke the services of another parser, the sub-parser must be first instantiated; the services of an instance are invoked by calling it using its **apply** method.

The following example shows a sub-parser invocation:

```
parser callee(packet_in packet, out IPv4 ipv4) { /* body omitted */ }
parser caller(packet_in packet, out Headers h) {
     callee() subparser;  // instance of callee
     state subroutine {
          subparser.apply(packet, h.ipv4);  // invoke sub-parser
          transition accept;  // accept if sub-parser ends in accept state
     }
}
```

The semantics of a sub-parser invocation can be described as follows:

- The state invoking the sub-parser is split into two half-states at the parser invocation statement.
- The top half includes a transition to the sub-parser `start` state.
- The sub-parser's `accept` state is identified with the bottom half of the current state
- The sub-parser's `reject` state is identified with the reject state of the current parser.

Figure 9 shows a diagram of this process.

Note that since P4 requires definitions to precede uses, it is impossible to create recursive (or mutually recursive) parsers.

When a parser is instantiated, local instantiations of stateful objects are evaluated recursively. That is, each instantiation of a parser has a unique set of local parser value sets, extern objects, inner parser instances, etc. Thus, in general, invoking a parser instance twice is not the same as invoking two copies of the same parser instance. Note however that local variables do not persist across invocations of the parser. This semantics also applies to direct invocation (see Section 15.1).

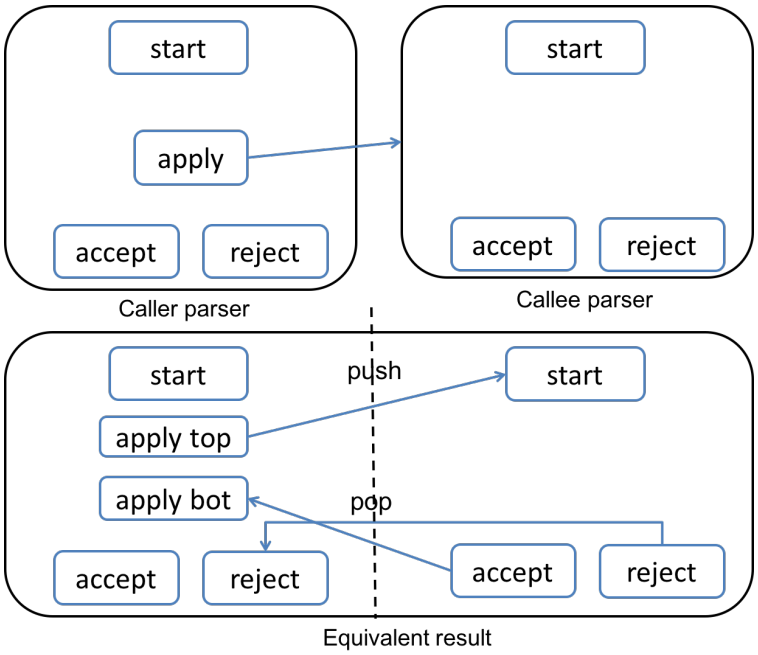

Figure 9. Semantics of invoking a sub-parser: top: original program, bottom: equivalent program.

Architectures may impose (static or dynamic) constraints on the number of parser states that can be traversed for processing each packet. For example, a compiler for a specific target may reject parsers containing loops that cannot be unrolled at compilation time or that may contain cycles that do not advance the cursor. If a parser aborts execution dynamically because it exceeded the time budget allocated for parsing, the parser should transition to `reject` and set the standard error **`error`**`.ParserTimeout`.

13.11. Parser 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.

Value sets are declared locally within a parser. They should be declared before being referenced in parser `keysetExpression` and can be used as a label in a select expression.

The syntax for declaring value sets is:

```
valueSetDeclaration
  : optAnnotations
      VALUESET "<" baseType ">" "(" expression ")" name ";"
  | optAnnotations
      VALUESET "<" tupleType ">" "(" expression ")" name ";"
  | optAnnotations
      VALUESET "<" typeName ">" "(" expression ")" name ";"
```

```
    ;
```

Parser Value Sets support a `size` argument to provide hints to the compiler to reserve hardware resources to implement the value set. For example, this parser value set:

```
value_set<bit<16>>(4) pvs;
```

creates a value_set of size 4 with entries of type `bit<16>`.

The semantics of the `size` argument is similar to the `size` property of a table. If a value set has a `size` argument with value `N`, it is recommended that a compiler should choose a data plane implementation that is capable of storing `N` value set entries. See "Size property of P4 tables and parser value sets" P4SizeProperty for further discussion on the implementation of parser value set size.

The value set is populated by the control plane by methods specified in the P4Runtime specification[4].

14. Control blocks

P4 parsers are responsible for extracting bits from a packet into headers. These headers (and other metadata) can be manipulated and transformed within `control` blocks. The body of a control block resembles a traditional imperative program. Within the body of a control block, match-action units can be invoked to perform data transformations. Match-action units are represented in P4 by constructs called `tables`.

Syntactically, a `control` block is declared with a name, parameters, optional type parameters, and a sequence of declarations of constants, variables, `action`s, `table`s, and other instantiations:

```
controlDeclaration
    : controlTypeDeclaration optConstructorParameters
      /* controlTypeDeclaration cannot contain type parameters */
      "{" controlLocalDeclarations APPLY controlBody "}"
    ;

controlLocalDeclarations
    : /* empty */
    | controlLocalDeclarations controlLocalDeclaration
    ;

controlLocalDeclaration
    : constantDeclaration
    | actionDeclaration
    | tableDeclaration
    | instantiation
    | variableDeclaration
    ;
```

[4]The P4Runtime API is defined as a Google Protocol Buffer `.proto` file and an accompanying English specification document here: https://github.com/p4lang/p4runtime

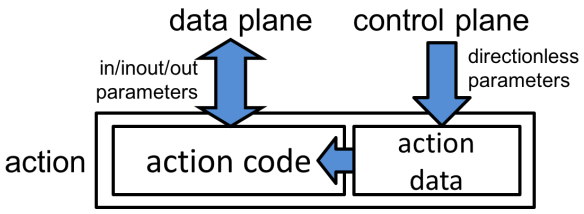

Figure 10. Actions contain code and data. The code is in the P4 program, while the data is provided in the table entries, typically populated by the control plane. Other parameters are bound by the data plane.

```
controlBody
    : blockStatement
    ;
```

It is illegal to instantiate a **parser** within a **control** block. For a description of the `optConstructorParameters`, which can be used to build parameterized control blocks, see Section 15.

Unlike control type declarations, control declarations may not be generic—e.g., the following declaration is illegal:

```
control C<H>(inout H data) { /* Body omitted */ }
```

P4 does not support exceptional control-flow within a **control** block. The only statement which has a non-local effect on control flow is **exit**, which causes execution of the enclosing control block to immediately terminate. That is, there is no equivalent of the **verify** statement or the `reject` state from parsers. Hence, all error handling must be performed explicitly by the programmer.

The rest of this section describes the core components of a **control** block, starting with actions.

14.1. Actions

Actions are code fragments that can read and write the data being processed. Actions may contain data values that can be written by the control plane and read by the data plane. Actions are the main construct by which the control plane can dynamically influence the behavior of the data plane. Figure 10 shows the abstract model of an **action**.

```
actionDeclaration
    : optAnnotations ACTION name "(" parameterList ")" blockStatement
    ;
```

Syntactically actions resemble functions with no return value. Actions may be declared within a control block; in this case they can only be used within instances of that control block.

The following example shows an action declaration:

```
action Forward_a(out bit<9> outputPort, bit<9> port) {
    outputPort = port;
```

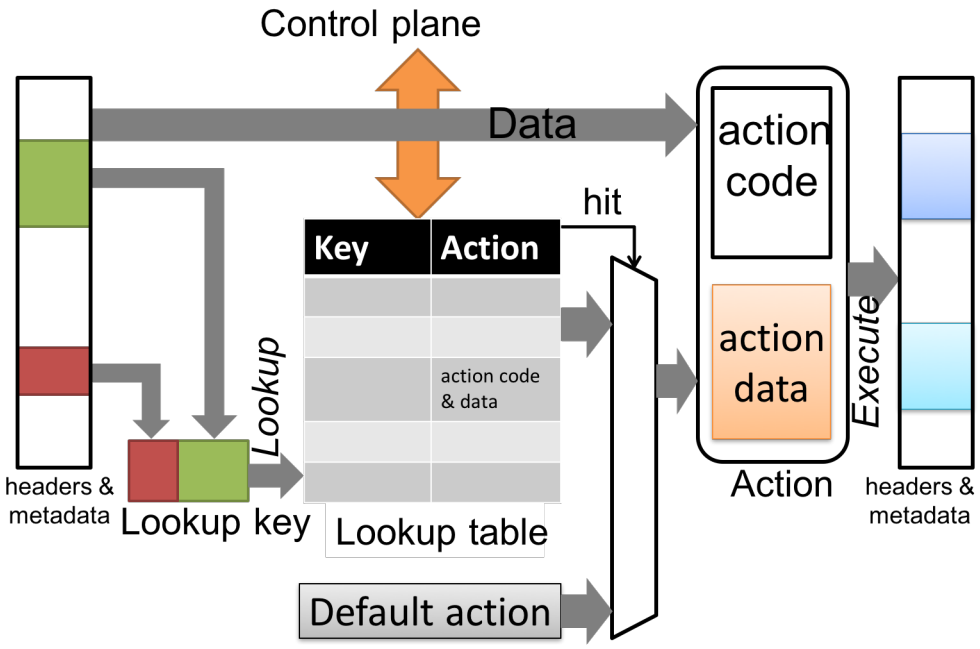

Figure 11. Match-Action Unit Dataflow.

```
}
```

Action parameters may not have **extern** types. Action parameters that have no direction (e.g., `port` in the previous example) indicate "action data." All such parameters must appear at the end of the parameter list. When used in a match-action table (see Section 14.2.1.2), these parameters will be provided by the table entries (e.g., as specified by the control plane, the `default_action` table property, or the **entries** table property).

The body of an action consists of a sequence of statements and declarations. No **table**, **control**, or **parser** applications can appear within actions.

Some targets may impose additional restrictions on action bodies—e.g., only allowing straight-line code, with no conditional statements or expressions.

14.1.1. Invoking actions

Actions can be executed in two ways:

- Implicitly: by tables during match-action processing.
- Explicitly: either from a **control** block or from another **action**. In either case, the values for all action parameters must be supplied explicitly, including values for the directionless parameters. In this case, the directionless parameters behave like **in** parameters.

14.2. Tables

A **table** describes a match-action unit. The structure of a match-action unit is shown in Figure 11. Processing a packet using a match-action table executes the following steps:

- Key construction.
- Key lookup in a lookup table (the "match" step). The result of key lookup is an "action".
- Action execution (the "action step") over the input data, resulting in mutations of the data.

A **table** declaration introduces a table instance. To obtain multiple instances of a table, it must be declared within a control block that is itself instantiated multiple times.

The look-up table is a finite map whose contents are manipulated asynchronously (read/write) by the target control plane, through a separate control-plane API (see Figure 11). Note that the term "table" is overloaded: it can refer to the P4 **table** objects that appear in P4 programs, as well as the internal look-up tables used in targets. We will use the term "match-action unit" when necessary to disambiguate.

Syntactically a table is defined in terms of a set of key-value properties. Some of these properties are "standard" properties, but the set of properties can be extended by target-specific compilers as needed. Note duplicated properties are invalid and the compiler should reject them.

```
tableDeclaration
    : optAnnotations TABLE name "{" tablePropertyList "}"
    ;

tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY '=' '{' keyElementList '}'
    | ACTIONS '=' '{' actionList '}'
    | optAnnotations optCONST ENTRIES '=' '{' entriesList '}'
    | optAnnotations optCONST nonTableKwName '=' initializer ';'
    ;

nonTableKwName
   : IDENTIFIER
   | TYPE_IDENTIFIER
   | APPLY
   | STATE
   | TYPE
   | PRIORITY
   ;
```

The standard table properties include:

- `key`: An expression that describes how the key used for look-up is computed.
- `actions`: A list of all actions that may be found in the table.

In addition, the tables may optionally define the following properties,

- `default_action`: an action to execute when the lookup in the lookup table fails to find a match for

the key used.
- `size`: an integer specifying the desired size of the table.
- **`entries`**: entries that are initially added to a table when the P4 program is loaded, some or all of which may be unchangeable by the control plane software.
- `largest_priority_wins` - Only useful for some tables with the **`entries`** property. See section 14.2.1.4 for details.
- `priority_delta` - Only useful for some tables with the **`entries`** property. See section 14.2.1.4 for details.

The compiler must set the `default_action` to `NoAction` (and also insert it into the list of `actions`) for tables that do not define the `default_action` property. `Hence, all tables can be thought of as having a` default_action` property, either implicitly or explicitly.

In addition, tables may contain architecture-specific properties (see Section 14.2.1.6).

A property marked as **`const`** cannot be changed dynamically by the control plane. The `key`, `actions`, and `size` properties cannot be modified so the **`const`** keyword is not needed for these.

14.2.1. Table properties

14.2.1.1. Keys The `key` is a table property which specifies the data-plane values that should be used to look up an entry. A key is a list of pairs of the form `(e : m)`, where `e` is an expression that describes the data to be matched in the table, and `m` is a **`match_kind`** that describes the algorithm used to perform the lookup (see Section 7.1.3).

```
keyElementList
    : /* empty */
    | keyElementList keyElement
    ;

keyElement
    : expression ":" name optAnnotations ";"
    ;
```

For example, consider the following program fragment:

```
table Fwd {
    key = {
       ipv4header.dstAddress : ternary;
       ipv4header.version    : exact;
    }
    // more fields omitted
}
```

Here the key comprises two fields from the `ipv4header` header: `dstAddress` and `version`. The **`match_kind`** elements serve three purposes:

- They specify the algorithm used to match data-plane values against the entries in the table at runtime.
- They are used to synthesize the control-plane API that is used to populate the table.

- They are used by the compiler back-end to allocate resources for the implementation of the table.

The P4 core library contains three predefined `match_kind` identifiers:

```
match_kind {
    exact,
    ternary,
    lpm
}
```

These identifiers correspond to the $P4_{14}$ match kinds with the same names. The semantics of these match kinds is actually not needed to describe the behavior of the P4 abstract machine; how they are used influences only the control-plane API and the implementation of the look-up table. From the point of view of the P4 program, a look-up table is an abstract finite map that is given a key and produces as a result either an action or a "miss" indication, as described in Section 14.2.3.

The expected meaning of these values is as follows:

- an `exact` match kind on a key field means that the value of the field in the table specifies exactly the value the lookup key field must have in order to match. This is applicable for all legal key fields whose types support equality comparisons.
- a `ternary` match kind on a key field means that the field in the table specifies a set of values for the key field using a value and a mask. The meaning of the `(value, mask)` pair is similar to the P4 mask expressions, as described in Section 8.15.3: a key field `k` matches the table entry when `k & mask == value & mask`.
- a `lpm` (longest prefix match) match kind on a key field is a specific type of `ternary` match where the mask is required to have a form in binary that is a contiguous set of 1 bits followed by a contiguous set of 0 bits. Masks with more 1 bits have automatically higher priorities. A mask with all bits 0 is legal.

Some table entries, in particular the ones with at least one `ternary` field, also require a priority value. A priority is a numeric value which is used to break ties when a particular key belongs to multiple sets. When table entries are specified in the P4 program the priorities are generated by the compiler; when entries are specified by the control-plane, the priority may need to be explicitly specified. Entries with higher priority are matched first. This specification does not mandate whether "higher" priorities are represented by higher or lower numeric values; this choice is left to the target implementation.

An example specifying entries for a table is given in Section 14.2.1.4.

If a table has no `key` property, or if the value of its `key` property is the empty tuple, i.e. `key = {}`, then it contains no look-up table, just a default action—i.e., the associated lookup table is always the empty map.

Each key element can have an optional `@name` annotation which is used to synthesize the control-plane-visible name for the key field.

Note some implementations might only support a limited number of keys or a limited combinations of match_kind for the keys. The implementation should reject those cases with an error message in this case.

14.2.1.2. Actions A table must declare all possible actions that may appear within the associated lookup table or in the default action. This is done with the `actions` property; the value of this property is always an `actionList`:

```
actionList
    : /* empty */
    | actionList optAnnotations actionRef ";"
    ;

actionRef
    : prefixedNonTypeName
    | prefixedNonTypeName "(" argumentList ")"
    ;
```

To illustrate, recall the example Very Simple Switch program in Section 5.3:

```
action Drop_action() {
  outCtrl.outputPort = DROP_PORT;
}

action Rewrite_smac(EthernetAddress sourceMac) {
  headers.ethernet.srcAddr = sourceMac;
}

table smac {
    key = { outCtrl.outputPort : exact; }
    actions = {
        Drop_action;
        Rewrite_smac;
    }
}
```

- The entries in the `smac` **`table`** may contain two different actions: `Drop_action` and `Rewrite_mac`.
- The `Rewrite_smac` action has one parameter, `sourceMac`, which in this case will be provided by the control plane.

Each action in the list of actions for a table must have a distinct name—e.g., the following program fragment is illegal:

```
action a() {}
control c() {
    action a() {}
    // Illegal table: two actions with the same name
    table t { actions = { a; .a; } }
}
```

Each action parameter that has a direction (**`in`**, **`inout`**, or **`out`**) must be bound in the `actions` list specifi-

cation; conversely, no directionless parameters may be bound in the list. The expressions supplied as arguments to an **action** are not evaluated until the action is invoked. Applying tables, whether directly via an expression like `table1.apply().hit`, or indirectly, are forbidden in the expressions supplied as action arguments.

```
action a(in bit<32> x) { /* body omitted */ }
bit<32> z;
action b(inout bit<32> x, bit<8> data) { /* body omitted */ }
table t {
    actions = {
       // a; -- illegal, x parameter must be bound
       a(5);  // binding a's parameter x to 5
       b(z);  // binding b's parameter x to z
       // b(z, 3); -- illegal, cannot bind directionless data parameter
       // b(); -- illegal, x parameter must be bound
       // a(table2.apply().hit ? 5 : 3); -- illegal, cannot apply a table here
    }
}
```

14.2.1.3. Default action The default action for a table is an action that is invoked automatically by the match-action unit whenever the lookup table does not find a match for the supplied key.

If present, the `default_action` property must appear after the **action** property. It may be declared as **const**, indicating that it cannot be changed dynamically by the control-plane. The **default action** must be one of the actions that appear in the actions list. In particular, the expressions passed as **in**, **out**, or **inout** parameters must be syntactically identical to the expressions used in one of the elements of the `actions` list.

For example, in the above **table** we could set the default action as follows (marking it also as constant):

```
const default_action = Rewrite_smac(48w0xAA_BB_CC_DD_EE_FF);
```

Note that the specified default action must supply arguments for the control-plane-bound parameters (i.e., the directionless parameters), since the action is synthesized at compilation time. The expressions supplied as arguments for parameters with a direction (**in**, **inout**, or **out**) are evaluated when the action is invoked while the expressions supplied as arguments for directionless parameters are evaluated at compile time.

Continuing the example from the previous section, the following are several legal and illegal specifications of default actions for the **table** `t`:

```
  default_action = a(5); // OK - no control-plane parameters
  // default_action = a(z); -- illegal, a's x parameter is already bound to 5
  default_action = b(z,8w8); // OK - bind b's data parameter to 8w8
  // default_action = b(z); -- illegal, b's data parameter is not bound
  // default_action = b(x, 3); -- illegal: x parameter of b bound to x instead of z
```

14.2.1.4. Entries While table entries are typically installed by the control plane, tables may also be initialized at compile time with a set of entries.

Declaring these entries with **const entries** is useful in situations where tables are used to implement fixed algorithms—defining table entries statically enables expressing these algorithms directly in P4, which allows the compiler to infer how the table is actually used and potentially make better allocation decisions for targets with limited resources.

Declaring entries with **entries** (without the **const** qualifier) enables one to specify a mix of some immutable entries that are always in the table, and some mutable entries that the control plane is allowed to later change or remove.

Entries declared in the P4 source are installed in the table when the program is loaded onto the target. Entries cannot be specified for a table with no key (see Sec. 14.2.1.1).

Table entries are defined using the following syntax:

```
tableProperty
 : optAnnotations optCONST ENTRIES '=' '{' entriesList '}'
 ;

entriesList
    : /* empty */
    | entriesList entry
    ;

optCONST
    : /* empty */
    | CONST
    ;

entryPriority
 : PRIORITY '=' INTEGER ":"
 | PRIORITY '=' '(' expression ')' ":"
 ;

entry
    : optCONST entryPriority keysetExpression ':' actionRef optAnnotations ';'
    | optCONST keysetExpression ':' actionRef optAnnotations ';'
    ;
```

Table entries defined using **const entries** are immutable—i.e., they can only be read by the control plane. The control plane is not allowed to remove or modify any entries defined within **const entries**, nor is it allowed to add entries to such a table. It is allowed for individual entries to have the **const** keyword before them, but this is redundant when the entries are declared using **const entries**.

Table entries defined using **entries** (without a **const** qualifier before it) may have **const** before them, or not, independently for each entry. Entries with **const** before them may not be modified or removed by the control plane. Entries without **const** may be modified or removed by the control plane. It is permitted for the control plane to add entries to such a table (subject to table capacity limitations), unlike tables declared with **const entries**.

Whether the control plane is allowed to modify a table's default action at run time is determined by the table's `default_action` table property (see Section 14.2.1.3), independently of whether the control plane is allowed to modify the entries of the table.

The `keysetExpression` component of an entry is a tuple that must provide a field for each key in the table keys (see Sec. 14.2.1). The table key type must match the type of the element of the set. The `actionRef` component must be an action which appears in the table actions list (and must not have the `@defaultonly` annotation), with all its arguments bound.

If no entry priorities are specified in the source code, and if the runtime API requires a priority for the entries of a table—e.g. when using the P4 Runtime API, tables with at least one `ternary` search key field—then the entries are matched in program order, stopping at the first matching entry. Architectures should define the significance of entry order (if any) for other kinds of tables.

Depending on the **`match_kind`** of the keys, key set expressions may define one or multiple entries. The compiler will synthesize the correct number of entries to be installed in the table. Target constraints may further restrict the ability of synthesizing entries. For example, if the number of synthesized entries exceeds the table size, the compiler implementation may choose to issue a warning or an error, depending on target capabilities.

To illustrate, consider the following example:

```
header hdr {
    bit<8>  e;
    bit<16> t;
    bit<8>  l;
    bit<8>  r;
    bit<1>  v;
}

struct Header_t {
    hdr h;
}
struct Meta_t {}

control ingress(inout Header_t h, inout Meta_t m,
                inout standard_metadata_t standard_meta) {

    action a() { standard_meta.egress_spec = 0; }
    action a_params(bit<9> x) { standard_meta.egress_spec = x; }

    table t_exact_ternary {

      key = {
            h.h.e : exact;
            h.h.t : ternary;
        }

    actions = {
            a;
```

```
            a_params;
        }

    default_action = a;

        const entries = {
            (0x01, 0x1111 &&& 0xF   ) : a_params(1);
            (0x02, 0x1181           ) : a_params(2);
            (0x03, 0x1111 &&& 0xF000) : a_params(3);
            (0x04, 0x1211 &&& 0x02F0) : a_params(4);
            (0x04, 0x1311 &&& 0x02F0) : a_params(5);
            (0x06, _                ) : a_params(6);
            _                         : a;
        }
    }

}
```

In this example we define a set of 7 entries, all of which invoke action `a_params` except for the final entry which invokes action `a`. Once the program is loaded, these entries are installed in the table in the order they are enumerated in the program.

Entry priorities If a table has fields where their **`match_kind`**s are all `exact` or `lpm`, there is no reason to assign numeric priorities to its entries. If they are all `exact`, duplicate keys are not allowed, and thus every lookup key can match at most one entry, so there is no need for a tiebreaker. If there is an `lpm` field, the priority of the entry corresponds to the length of the prefix, i.e. if a lookup key matches multiple prefixes, the longest prefix is always the winner.

For tables with other **`match_kind`** values, e.g. at least one `ternary` field, in general it is possible to install multiple entries such that the same lookup key can match the key of multiple entries installed into the table at the same time. Control plane APIs such as P4Runtime API[4] and TDI[5] require control plane software to provide a numeric priority with each entry added to such a table. This enables the data plane to determine which of several matching entries is the "winner", i.e. the one entry whose action is invoked.

Unfortunately there are two commonly used, but different, ways of interpreting numeric priority values.

The P4Runtime API requires numeric priorities to be positive integers, i.e. 1 or larger, and defines that entries with larger priorities must win over entries with smaller priorities. We will call this convention `largest_priority_wins`.

TDI requires numeric priorities to be non-negative integers, i.e. 0 or larger, and defines that entries with smaller priorities must win over entries with larger priorities. We will call this convention `smallest_priority_wins`.

We wish to support either of these conventions when developers specify priorities for initial table entries in the program. Thus there is a table property `largest_priority_wins`. If explicitly specified for a

[4]The P4Runtime API is defined as a Google Protocol Buffer `.proto` file and an accompanying English specification document here: https://github.com/p4lang/p4runtime

[5]TDI is the Table Driven Interface. More information can be found here: https://github.com/p4lang/tdi

table, its value must be boolean. If **true**, then the priority values use the `largest_priority_wins` convention. If **false**, then the priority values use the `smallest_priority_wins` convention. If the table property is not present at all, then the default convention is **true**, corresponding to `largest_priority_wins`.

We also wish to support developers that want the convenience of predictable entry priority values automatically selected by the compiler, without having to write them in the program, plus the ability to specify entry priorities explicitly, if they wish.

In some cases, developers may wish the initial priority values to have "gaps" between their values, to leave room for possible later insertion of new entries between two initial entries. They can achieve this by explicitly specifying all priority values, of course, but as a convenience we define the table property `priority_delta` to be a positive integer value, with a default value of 1 if not specified for a table, to use as a default difference between the priorities of consecutive entries.

There are two steps that occur at compile time for a table with the **entries** property involving entry priorities:

- Determine the value of the priority of every entry in the **entries** list.
- Issue any errors or warnings that are appropriate for these priority values. Warnings may be suppressed via an appropriate `@noWarn` annotation.

These steps are performed independently for each table with the **entries** property, and each is described in more detail below.

In general, if the developer specifies a priority value for an entry, that is the value that will be used.

If the developer does not specify priority values for any entry, then the compiler calculates priority values for every entry as follows:

```
// For this pseudocode, table entries in the `entries` list are
// numbered 0 through n-1, 0 being the first to appear in order in the
// source code.  Their priority values are named prio[0] through
// prio[n-1].
int p = 1;
if (largest_priority_wins == true) {
    for (int j = n-1; j >= 0; j -= 1) {
        prio[j] = p;
        p += priority_delta;
    }
} else {
    for (int j = 0; j < n; j += 1) {
        prio[j] = p;
        p += priority_delta;
    }
}
```

If the developer specifies priority values for at least one entry, then in order to simplify the rules for determining priorities of entries without one in the source code, the first entry must have a priority value explicitly provided. The priorities of entries that do not have one in the source code (if any) are determined as follows:

```
// Same conventions here as in the previous block of pseudocode above.
// If entry j has a priority value specified in the source code,
// prio_specified[j] is true, otherwise it is false.
assert(prio_specified[0]);  // compile time error if prio_specified[0] is false
p = prio[0];
for (int j = 1; j < n; j += 1) {
    if (prio_specified[j]) {
        p = prio[j];
    } else {
        if (largest_priority_wins == true) {
            p -= priority_delta;
        } else {
            p += priority_delta;
        }
        prio[j] = p;
    }
}
```

This is the end of the first step: determining entry priorities.

The priorities determined in this way are the values used when the P4 program is first loaded into a device. Afterwards, the priorities may only change by means provided by the control plane API in use.

In the second step, the compiler issues errors for out of range priority values, and/or warnings for certain combinations of entry priorities that might be unintended by the developer, unless the developer explicitly disables those warnings.

If any priority values are negative, or larger than the maximum supported value, that is a compile time error.

If the annotation `@noWarn("duplicate_priorities")` is not used on the **`entries`** table property, then the compiler issues a warning if any two entries for the same table have equal priority values. Both P4Runtime and TDI leave it unspecified which entry is the winner if a lookup key matches multiple keys that all have the same priority, hence a warning is useful to less experienced developers that are unfamiliar with this unspecified behavior.

If the annotation `@noWarn("duplicate_priorities")` is used on the **`entries`** table property, then no warnings of this type are ever issued by the compiler. Using equal priority values for multiple entries in the same table is sometimes useful in reducing the number of hardware updates required when adding entries to such a table.

If the annotation `@noWarn("entries_out_of_priority_order")` is not used on the **`entries`** table property, then the compiler issues a warning if:

- If `largest_priority_wins` is **`true`** for the table, and there is any pair of consecutive entries where `prio[j] < prio[j+1]`, then a warning is issued for that pair of entries.
- If `largest_priority_wins` is **`false`** for the table, and there is any pair of consecutive entries where `prio[j] > prio[j+1]`, then a warning is issued for that pair of entries.

This warning is useful to developers that want the order that entries appear in the source code to match the relative priority of entries in the target device.

If the annotation `@noWarn("entries_out_of_priority_order")` is used on the **`entries`** table property, then no warnings of this type are ever issued by the compiler for this table. This option is provided

for developers who explicitly choose to specify entries in an order that does not match their relative priority order.

The following example is the same as the first example in section 14.2.1.4, except for the definition of table `t_exact_ternary` shown below.

```
table t_exact_ternary {
    key = {
        h.h.e : exact;
        h.h.t : ternary;
    }

    actions = {
        a;
        a_params;
    }

    default_action = a;

    largest_priority_wins = false;
    priority_delta = 10;
    @noWarn("duplicate_priorities")
    entries = {
        const priority=10: (0x01, 0x1111 &&& 0xF   ) : a_params(1);
                           (0x02, 0x1181           ) : a_params(2); // priority=20
                           (0x03, 0x1000 &&& 0xF000) : a_params(3); // priority=30
        const              (0x04, 0x0210 &&& 0x02F0) : a_params(4); // priority=40
              priority=40: (0x04, 0x0010 &&& 0x02F0) : a_params(5);
                           (0x06, _                ) : a_params(6); // priority=50
    }
}
```

The entries that do not have an explicit priority specified will be assigned the priority values shown in the comments, because `priority_delta` is 10, and because of those entries that do have priority values specified.

Normally this program would cause a warning about multiple entries with the same priority of 40, but those warnings will be suppressed because of the `@noWarn("duplicate_priorities")` annotation.

14.2.1.5. Size The `size` is an optional property of a table. When present, its value must always be a compile-time known value that is an integer. The `size` property is specified in units of number of table entries.

If a table is specified with a `size` property of value `N`, it is recommended that a compiler should choose a data plane implementation that is capable of storing `N` table entries. This does not guarantee that an arbitrary set of `N` entries can always be inserted in such a table, only that there is some set of `N` entries that can be inserted. For example, attempts to add some combinations of `N` entries may fail because the compiler selected a hash table with `O(1)` guaranteed search time. See "Size property of P4 tables and parser value sets" P4SizeProperty for further discussion on some P4 table implementations

and what they are able to guarantee.
If a P4 implementation must dimension table resources at compile time, they may treat it as an error if they encounter a table with no `size` property.

Some P4 implementations may be able to dynamically dimension table resources at run time. If a `size` value is specified in the P4 program, it is recommended that such an implementation uses the `size` value as the initial capacity of the table.

14.2.1.6. Additional properties

A **`table`** declaration defines its essential control and data plane interfaces—i.e., keys and actions. However, the best way to implement a table may actually depend on the nature of the entries that will be installed at runtime (for example, tables could be dense or sparse, could be implemented as hash-tables, associative memories, tries, etc.) In addition, some architectures may support extra table properties whose semantics lies outside the scope of this specification. For example, in architectures where table resources are statically allocated, programmers may be required to define a `size` table property, which can be used by the compiler back-end to allocate storage resources. However, these architecture-specific properties may not change the semantics of table lookups, which always produce either a `hit` and an action or a `miss`—they can only change how those results are interpreted on the state of the data plane. This restriction is needed to ensure that it is possible to reason about the behavior of tables during compilation.

As another example, an `implementation` property could be used to pass additional information to the compiler back-end. The value of this property could be an instance of an **`extern`** block chosen from a suitable library of components. For example, the core functionality of the $P4_{14}$ table `action_profile` constructs could be implemented on architectures that support this feature using a construct such as the following:

```
extern ActionProfile {
   ActionProfile(bit<32> size); // number of distinct actions expected
}
table t {
    key = { /* body omitted */ }
    size = 1024;
    implementation = ActionProfile(32);  // constructor invocation
}
```

Here the action profile might be used to optimize for the case where the table has a large number of entries, but the actions associated with those entries are expected to range over a small number of distinct values. Introducing a layer of indirection enables sharing identical entries, which can significantly reduce the table's storage requirements.

14.2.2. Match-action unit invocation

A **`table`** can be invoked by calling its **`apply`** method. Calling an apply method on a table instance returns a value with a **`struct`** type with three fields. This structure is synthesized by the compiler automatically. For each **`table`** `T`, the compiler synthesizes an **`enum`** and a **`struct`**, shown in pseudo-P4:

```
enum action_list(T) {
   // one field for each action in the actions list of table T
```

```
}
struct apply_result(T) {
    bool hit;
    bool miss;
    action_list(T) action_run;
}
```

The evaluation of the `apply` method sets the `hit` field to `true` and the field `miss` to `false` if a match is found in the lookup-table; if a match is not found `hit` is set to `false` and `miss` to `true`. These bits can be used to drive the execution of the control-flow in the control block that invoked the table:

```
if (ipv4_match.apply().hit) {
    // there was a hit
} else {
    // there was a miss
}

if (ipv4_host.apply().miss) {
    ipv4_lpm.apply(); // Look up the route only if host table missed
}
```

The `action_run` field indicates which kind of action was executed (irrespective of whether it was a hit or a miss). It can be used in a switch statement:

```
switch (dmac.apply().action_run) {
    Drop_action: { return; }
}
```

14.2.3. Match-action unit execution semantics

The semantics of a table invocation statement:

```
m.apply();
```

is given by the following pseudocode (see also Figure 11):

```
apply_result(m) m.apply() {
    apply_result(m) result;

    var lookupKey = m.buildKey(m.key); // using key block
    action RA = m.table.lookup(lookupKey);
    if (RA == null) {      // miss in lookup table
       result.hit = false;
       RA = m.default_action;  // use default action
    }
    else {
```

```
        result.hit = true;
    }
    result.miss = !result.hit;
    result.action_run = action_type(RA);
    evaluate_and_copy_in_RA_args(RA);
    execute(RA);
    copy_out_RA_args(RA);
    return result;
}
```

The behavior of the `buildKey` call in the pseudocode above is to evaluate each key expression in the order they appear in the table key definition. The behavior must be the same as if the result of evaluating each key expression is assigned to a fresh temporary variable, before starting the evaluation of the following key expression. For example, this P4 table definition and apply call:

```
bit<8> f1 (in bit<8> a, inout bit<8> b) {
    b = a + 5;
    return a >> 1;
}
bit<8> x;
bit<8> y;
table t1 {
    key = {
        y & 0x7  : exact @name("masked_y");
        f1(x, y) : exact @name("f1");
        y        : exact;
    }
    // ... rest of table properties defined here, not relevant to example
}
apply {
    // assign values to x and y here, not relevant to example
    t1.apply();
}
```

is equivalent in behavior to the following table definition and apply call:

```
// same definition of f1, x, and y as before, so they are not repeated here
bit<8> tmp_1;
bit<8> tmp_2;
bit<8> tmp_3;
table t1 {
    key = {
        tmp_1 : exact @name("masked_y");
        tmp_2 : exact @name("f1");
        tmp_3 : exact @name("y");
    }
```

```
    // ... rest of table properties defined here, not relevant to example
}
apply {
    // assign values to x and y here, not relevant to example
    tmp_1 = y & 0x7;
    tmp_2 = f1(x, y);
    tmp_3 = y;
    t1.apply();
}
```

Note that the second code example above is given in order to specify the behavior of the first one. An implementation is free to choose any technique that achieves this behavior[6].

14.3. The Match-Action Pipeline Abstract Machine

We can describe the computational model of a match-action pipeline, embodied by a control block: the body of the control block is executed, similarly to the execution of a traditional imperative program:

- At runtime, statements within a block are executed in the order they appear in the control block.
- Execution of the **return** statement causes immediate termination of the execution of the current **control** block, and a return to the caller.
- Execution of the **exit** statement causes the immediate termination of the execution of the current **control** block and of all the enclosing caller **control** blocks.
- Applying a **table** executes the corresponding match-action unit, as described above.

14.4. Invoking controls

P4 allows controls to invoke the services of other controls, similar to subroutines. To invoke the services of another control, it must be first instantiated; the services of an instance are invoked by calling it using its **apply** method.

The following example shows a control invocation:

```
control Callee(inout IPv4 ipv4) { /* body omitted */ }
control Caller(inout Headers h) {
     Callee() instance;  // instance of callee
     apply {
          instance.apply(h.ipv4);  // invoke control
     }
}
```

As with parsers, when a control is instantiated, local instantiations of stateful objects are evaluated recursively. That is, each instantiation of a control has a unique set of local tables, extern objects, inner

[6] Most existing $P4_{16}$ programs today do not use function or method calls in table key expressions, and the order of evaluation of these key expressions makes no difference in the resulting lookup key value. In this overwhelmingly common case, if an implementation chooses to insert extra assignment statements to implement side-effecting key expressions, but does not insert them when there are no side-effecting key expressions, then in typical programs they will almost never be inserted.

control instances, etc. Thus, in general, invoking a control instance twice is not the same as invoking two copies of the same control instance. Note however, that local variables do not persist across invocations of the control. This semantics also applies to direct invocation (see Section 15.1).

When a control is instantiated, all its local declarations of stateful instantiations are evaluated recursively. Each instantiation of a control will have a unique set of local tables, extern objects, and inner control instances. Thus, invoking a control instance twice is different from invoking two control instances each once, where the former accesses the same local stateful constructs while the latter access two different copies.

The exactly-once evaluation only applies to local stateful instantiations. For local variable declarations, whether in the **apply** block or out, and whether with initializers or not, they are always evaluated when a control instance is invoked. That is, local variables in a control never persist across invocations. For variables declared outside the **apply** block, they are evaluated at the beginning of execution.

All the behavior above also applies to direct invocation (see Section 15.1).

15. Parameterization

In order to support libraries of useful P4 components, both **parser**s and **control** blocks can be additionally parameterized through the use of constructor parameters.

Consider again the parser declaration syntax:

```
parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      "{" parserLocalElements parserStates "}"
    ;

optConstructorParameters
    : /* empty */
    | "(" parameterList ")"
    ;
```

From this grammar fragment we infer that a **parser** declaration may have two sets of parameters:

- The runtime parser parameters (`parameterList`)
- Optional parser constructor parameters (`optConstructorParameters`)

Constructor parameters must be directionless (i.e., they cannot be **in**, **out**, or **inout**) and where the parser is instantiated, it must be possible to fully evaluate the expressions supplied for these parameters at compilation time.

Consider the following example:

```
parser GenericParser(packet_in b, out Packet_header p)
                     (bool udpSupport) {   // constructor parameters
    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            16w0x0800: ipv4;
```

```
        }
    }
    state ipv4 {
        b.extract(p.ipv4);
        transition select(p.ipv4.protocol) {
           6: tcp;
           17: tryudp;
        }
    }
    state tryudp {
        transition select(udpSupport) {
            false: accept;
            true : udp;
        }
    }
    state udp {
         // body omitted
    }
}
```

When instantiating the `GenericParser` it is necessary to supply a value for the `udpSupport` parameter, as in the following example:

```
// topParser is a GenericParser where udpSupport = false
GenericParser(false) topParser;
```

15.1. Direct type invocation

Controls and parsers are often instantiated exactly once. As a light syntactic sugar, control and parser declarations with no constructor parameters may be applied directly, as if they were an instance. This has the effect of creating and applying a local instance of that type.

```
control Callee(/* parameters omitted */) { /* body omitted */ }

control Caller(/* parameters omitted */)(/* parameters omitted */) {
    apply {
        Callee.apply(/* arguments omitted */); // Callee is treated as an instance
    }
}
```

The definition of `Caller` is equivalent to the following.

```
control Caller(/* parameters omitted */)(/* parameters omitted */) {
    @name("Callee") Callee() Callee_inst; // local instance of Callee
    apply {
        Callee_inst.apply(/* arguments omitted */);         // Callee_inst is applied
```

```
    }
}
```

```
directApplication
    : typeName "." APPLY "(" argumentList ")" ";"
    | specializedType "." APPLY "(" argumentList ")" ";"
    ;
```

This feature is intended to streamline the common case where a type is instantiated exactly once.

The second production in the grammar allows direct calls for generic controls or parsers:

```
control Callee<T>(/* parameters omitted */) { /* body omitted */ }

control Caller(/* parameters omitted */)(/* parameters omitted */) {
    apply {
        Callee<bit<32>>.apply(/* arguments omitted */); // Callee<bit<32>> is treated as an in-
stance
    }
}
```

For completeness, the behavior of directly invoking the same type more than once is defined as follows.

- Direct type invocation in different scopes will result in different local instances with different fully-qualified control names.
- In the same scope, direct type invocation will result in a different local instance per invocation—however, instances of the same type will share the same global name, via the `@name` annotation. If the type contains controllable entities, then invoking it directly more than once in the same scope is illegal, because it will produce multiple controllable entities with the same fully-qualified control name.

See Section 18.3.2 for details of `@name` annotations.

No direct invocation is possible for controls or parsers that require constructor arguments. These need to be instantiated before they are invoked.

16. Deparsing

The inverse of parsing is deparsing, or packet construction. P4 does not provide a separate language for packet deparsing; deparsing is done in a **`control`** block that has at least one parameter of type `packet_out`.

For example, the following code sequence writes first an Ethernet header and then an IPv4 header into a `packet_out`:

```
control TopDeparser(inout Parsed_packet p, packet_out b) {
    apply {
        b.emit(p.ethernet);
        b.emit(p.ip);
```

```
    }
}
```

Emitting a header appends the header to the `packet_out` only if the header is valid. Emitting a header stack will emit all elements of the stack in order of increasing indices.

16.1. Data insertion into packets

The `packet_out` datatype is defined in the P4 core library, and reproduced below. It provides a method for appending data to an output packet called `emit`:

```
extern packet_out {
    void emit<T>(in T data);
}
```

The `emit` method supports appending the data contained in a header, header stack, **`struct`**, or header union to the output packet.

- When applied to a header, `emit` appends the data in the header to the packet if it is valid and otherwise behaves like a no-op.
- When applied to a header stack, `emit` recursively invokes itself to each element of the stack.
- When applied to a **`struct`** or header union, `emit` recursively invokes itself to each field. Note, a **`struct`** must not contain a field of type **`error`** or **`enum`** because these types cannot be serialized.

It is illegal to invoke `emit` on an expression whose type is a base type, **`enum`**, or **`error`**.

We can define the meaning of the `emit` method in pseudocode as follows:

```
packet_out {
    byte[] data;
    unsigned lengthInBits;
    void initializeForWriting() {
        this.data.clear();
        this.lengthInBits = 0;
    }
    /// Append data to the packet. Type T must be a header, header
    /// stack, header union, or struct formed recursively from those types
    void emit<T>(T data) {
        if (isHeader(T))
            if(data.valid$) {
                this.data.append(data);
                this.lengthInBits += data.lengthInBits;
            }
        else if (isHeaderStack(T))
            for (e : data)
                 emit(e);
        else if (isHeaderUnion(T) || isStruct(T))
            for (f : data.fields$)
```

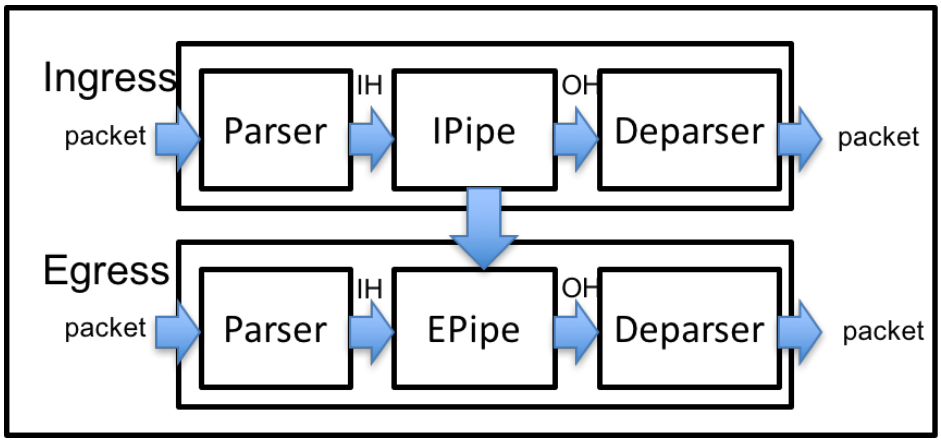

Figure 12. Fragment of example switch architecture.

```
                emit(e.f)
        // Other cases for T are illegal
    }
```

Here we use the special `valid$` identifier to indicate the hidden valid bit of headers and `fields$` to indicate the list of fields for a struct or header union. We also use standard `for-each` notation to iterate through the elements of a stack `(e : data)` and list of fields for header unions and structs `(f : data.fields$)`. The iteration order for a struct is the order those fields appear in the type declaration.

17. Architecture description

The architecture description must be provided by the target manufacturer in the form of a library P4 source file that contains at least one declaration for a **package**; this **package** must be instantiated by the user to construct a program for a target. For an example see the Very Simple Switch declaration from Section 5.1.

The architecture description file may pre-define data types, constants, helper package implementations, and errors. It must also declare the types of all the programmable blocks that will appear in the final target: **parser**s and **control** blocks. The programmable blocks may optionally be grouped together in packages, which can be nested.

Since some of the target components may manipulate user-defined types, which are unknown at the target declaration time, these are described using type variables, which must be used parametrically in the program—i.e., type variables are checked similar to Java generics, not C++ templates.

17.1. Example architecture description

The following example describes a switch by using two packages, each containing a parser, a match-action pipeline, and a deparser:

```
parser Parser<IH>(packet_in b, out IH parsedHeaders);
// ingress match-action pipeline
```

```
control IPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         out OH outputHeaders,
                         out T toEgress,
                         out OutControl outCtrl);
// egress match-action pipeline
control EPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         in T fromIngress,
                         out OH outputHeaders,
                         out OutControl outCtrl);
control Deparser<OH>(in OH outputHeaders, packet_out b);
package Ingress<T, IH, OH>(Parser<IH> p,
                           IPipe<T, IH, OH> map,
                           Deparser<OH> d);
package Egress<T, IH, OH>(Parser<IH> p,
                          EPipe<T, IH, OH> map,
                          Deparser<OH> d);
package Switch<T>(Ingress<T, _, _> ingress, Egress<T, _, _> egress);
```

Just from these declarations, even without reading a precise description of the target, the programmer can infer some useful information about the architecture of the described switch, as shown in Figure 12:

- The switch contains two separate **package**s `Ingress` and `Egress`.
- The `Parser`, `IPipe`, and `Deparser` in the `Ingress` package are chained together in order. In addition, the `Ingress.IPipe` block has an input of type `Ingress.IH`, which is an output of the `Ingress.Parser`.
- Similarly, the `Parser`, `EPipe`, and `Deparser` are chained in the `Egress` package.
- The `Ingress.IPipe` is connected to the `Egress.EPipe`, because the first outputs a value of type `T`, which is an input to the second. Note that the occurrences of the type variable `T` are instantiated with the same type in `Switch`. In contrast, the `Ingress` type `IH` and the `Egress` type `IH` may be different. To force them to be the same, we could instead declare `IH` and `OH` at the switch level: **package** `Switch<T,IH,OH>(Ingress<T, IH, OH> ingress, Egress<T, IH, OH> egress)`.

Hence, this architecture models a target switch that contains two separate channels between the ingress and egress pipeline:

- A channel that can pass data directly via its argument of type `T`. On a software target with shared memory between ingress and egress this could be implemented by passing directly a pointer; on an architecture without shared memory presumably the compiler will need to automatically synthesize serialization code.
- A channel that can pass data indirectly using a parser and deparser that serializes data into a packet and back.

17.2. Example architecture program

To construct a program for the architecture, the P4 program must instantiate a top-level **package** by passing values for all its arguments creating a variable called `main` in the top-level namespace. The types of the arguments must match the types of the parameters—after a suitable substitution of the

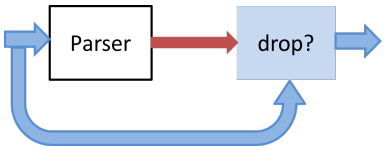

Figure 13. A packet filter target model. The parser computes a Boolean value, which is used to decide whether the packet is dropped.

type variables. The type substitution can be expressed directly, using type specialization, or can be inferred by a compiler, using a unification algorithm like Hindley-Milner.

For example, given the following type declarations:

```
parser Prs<T>(packet_in b, out T result);
control Pipe<T>(in T data);
package Switch<T>(Prs<T> p, Pipe<T> map);
```

and the following declarations:

```
parser P(packet_in b, out bit<32> index) { /* body omitted */ }
control Pipe1(in bit<32> data) { /* body omitted */ }
control Pipe2(in bit<8> data) { /* body omitted */ }
```

The following is a legal declaration for the top-level target:

```
Switch(P(), Pipe1()) main;
```

And the following is illegal:

```
Switch(P(), Pipe2()) main;
```

The latter declaration is incorrect because the parser `P` requires `T` to be `bit<32>`, while `Pipe2` requires `T` to be `bit<8>`.

The user can also explicitly specify values for the type variables (otherwise the compiler has to infer values for these type variables):

```
Switch<bit<32>>(P(), Pipe1()) main;
```

17.3. A Packet Filter Model

To illustrate the versatility of the P4 architecture description language, we give an example of another architecture: one which models a packet filter that makes a drop/no drop decision based only on the computation in a P4 parser, as shown in Figure 13.

This model could be used to program packet filters running in the Linux kernel. For example, we could replace the `tcpdump` language with the much more powerful P4 language; P4 can seamlessly support new protocols, while providing complete "type safety" during packet processing. For such a target,

the P4 compiler could generate an eBPF (Extended Berkeley Packet Filter) program, which is injected by the `tcpdump` utility into the Linux kernel, and executed by the eBPF kernel JIT compiler/runtime.

In this case the target is the Linux kernel, and the architecture model is a packet filter.

The declaration for this architecture is as follows:

```
parser Parser<H>(packet_in packet, out H headers);
control Filter<H>(inout H headers, out bool accept);

package Program<H>(Parser<H> p, Filter<H> f);
```

18. P4 abstract machine: Evaluation

The evaluation of a P4 program is done in two stages:

- static evaluation: at compile time the P4 program is analyzed and all stateful blocks are instantiated.
- dynamic evaluation: at runtime each P4 functional block is executed to completion, in isolation, when it receives control from the architecture

18.1. Compile-time known and local compile-time known values

Certain expressions in a P4 program have the property that their value can be determined at compile time. Moreover, for some of these expressions, their value can be determined only using information in the current scope. We call these compile-time known values and local compile-time known values respectively.

The following are local compile-time known values:

- Integer literals, Boolean literals, and string literals.
- Identifiers declared in an **`error`**, **`enum`**, or **`match_kind`** declaration.
- The **`default`** identifier.
- The `size` field of a value with type header stack.
- The _ identifier when used as a **`select`** expression label
- The expression `{#}` representing an invalid header or header union value.
- Instances constructed by instance declarations (Section 11.3) and constructor invocations.
- Identifiers that represent declared types, actions, functions, tables, parsers, controls, or packages.
- Tuple expression where all components are local compile-time known values.
- Structure-valued expressions, where all fields are local compile-time known values.
- Expressions evaluating to a list type, where all elements are local compile-time known values.
- Legal casts applied to local compile-time known values.
- The following expressions (`+`, `-`, `|+|`, `|-|`, `*`, `/` , `%`, `!`, `&`, `|`, `^`, `&&`, `||`, `<<` , `>>`, `~`, `/`, `>`, `<`, `==`, `!=`, `<=`, `>=`, `++`, `[:]`, `?:`) when their operands are all local compile-time known values.
- Expressions of the form `e.minSizeInBits()`, `e.minSizeInBytes()`, `e.maxSizeInBits()` and `e.maxSizeInBytes()` where the type of `e` is not generic.

The following are compile-time known values:

- All local compile-time known values.

- Constructor parameters (i.e., the declared parameters for a **parser**, **control**, etc.)
- Identifiers declared as constants using the **const** keyword.
- Tuple expression where all components are compile-time known values.
- Expressions evaluating to a list type, where all elements are compile-time known values.
- Structure-valued expressions, where all fields are compile-time known values.
- Expressions evaluating to a list type, where all elements are compile-time known values.
- Legal casts applied to compile-time known values.
- The following expressions (`+`, `-`, `|+|`, `|-|`, `*`, `/`, `%`, `cast`, `!`, `&`, `|`, `^`, `&&`, `||`, `<<`, `>>`, `~`, `/`, `>`, `<`, `==`, `!=`, `<=`, `>=`, `++`, `[:]`, `?:`) when their operands are all compile-time known values.
- Expressions of the form `e.minSizeInBits()`, `e.minSizeInBytes()`, `e.maxSizeInBits()` and `e.maxSizeInBytes()` where the the type of `e` is generic.

Intuitively, the main difference between compile-time known values and local compile-time known values is that the former also contains constructor parameters. The distinction is important when it comes to defining the meaning of features like types. For example, in the type **bit**<e>, the expression `e` must be a local compile-time known value. Suppose instead that `e` were a constructor parameter—i.e., merely a compile-time known value. In this situation, it would be impossible to resolve **bit**<e> to a concrete type using purely local information—we would have to wait until the constructor was instantiated and the value of `e` known.

18.2. Compile-time Evaluation

Evaluation of a program proceeds in order of declarations, starting in the top-level namespace:

- All declarations (e.g., `parsers`, `controls`, types, constants) evaluate to themselves.
- Each **table** evaluates to a table instance.
- Constructor invocations evaluate to stateful objects of the corresponding type. For this purpose, all constructor arguments are evaluated recursively and bound to the constructor parameters. Constructor arguments must be compile-time known values. The order of evaluation of the constructor arguments should be unimportant — all evaluation orders should produce the same results.
- Instantiations evaluate to named stateful objects.
- The instantiation of a **parser** or **control** block recursively evaluates all stateful instantiations declared in the block.
- The result of the program's evaluation is the value of the top-level `main` variable.

Note that all stateful values are instantiated at compilation time.

As an example, consider the following program fragment:

```
// architecture declaration
parser P(/* parameters omitted */);
control C(/* parameters omitted */);
control D(/* parameters omitted */);

package Switch(P prs, C ctrl, D dep);

extern Checksum16 { /* body omitted */}
```

```
// user code
Checksum16() ck16; // checksum unit instance

parser TopParser(/* parameters omitted */)(Checksum16 unit) { /* body omitted */}
control Pipe(/* parameters omitted */) { /* body omitted */}
control TopDeparser(/* parameters omitted */)(Checksum16 unit) { /* body omitted */}

Switch(TopParser(ck16),
       Pipe(),
       TopDeparser(ck16)) main;
```

The evaluation of this program proceeds as follows:

1. The declarations of `P`, `C`, `D`, `Switch`, and `Checksum16` all evaluate to themselves.
2. The `Checksum16() ck16` instantiation is evaluated and it produces an object named `ck16` with type `Checksum16`.
3. The declarations for `TopParser`, `Pipe`, and `TopDeparser` evaluate as themselves.
4. The `main` variable instantiation is evaluated:
   (a) The arguments to the constructor are evaluated recursively
   (b) `TopParser(ck16)` is a constructor invocation
   (c) Its argument is evaluated recursively; it evaluates to the `ck16` object
   (d) The constructor itself is evaluated, leading to the instantiation of an object of type `TopParser`
   (e) Similarly, `Pipe()` and `TopDeparser(ck16)` are evaluated as constructor calls.
   (f) All the arguments of the `Switch` package constructor have been evaluated (they are an instance of `TopParser`, an instance of `Pipe`, and an instance of `TopDeparser`). Their signatures are matched with the `Switch` declaration.
   (g) Finally, the `Switch` constructor can be evaluated. The result is an instance of the `Switch` package (that contains a `TopParser` named `prs` the first parameter of the `Switch`; a `Pipe` named `ctrl`; and a `TopDeparser` named `dep`).
5. The result of the program evaluation is the value of the `main` variable, which is the above instance of the `Switch` **package**.

Figure 14 shows the result of the evaluation in a graphical form. The result is always a graph of instances. There is only one instance of `Checksum16`, called `ck16`, shared between the `TopParser` and `TopDeparser`. Whether this is possible is architecture-dependent. Specific target compilers may require distinct checksum units to be used in distinct blocks.

18.3. Control plane names

Every controllable entity exposed in a P4 program must be assigned a unique, fully-qualified name, which the control plane may use to interact with that entity. The following entities are controllable.

- value sets
- tables
- keys
- actions

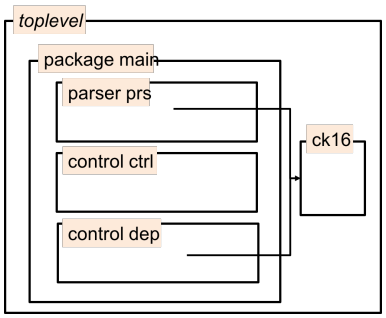

Figure 14. Evaluation result.

- extern instances

A fully qualified name consists of the local name of a controllable entity prepended with the fully qualified name of its enclosing namespace. Hence, the following program constructs, which enclose controllable entities, must themselves have unique, fully-qualified names.

- control instances
- parser instances

Evaluation may create multiple instances from one type, each of which must have a unique, fully-qualified name.

18.3.1. Computing control-plane names

The fully-qualified name of a construct is derived by concatenating the fully-qualified name of its enclosing construct with its local name. Constructs with no enclosing namespace, i.e. those defined at the global scope, have the same local and fully-qualified names. The local names of controllable entities and enclosing constructs are derived from the syntax of a P4 program as follows.

18.3.1.1. Value sets For each `value_set` construct, its syntactic name becomes the local name of the value set. For example:

```
struct vsk_t {
    @match(ternary)
    bit<16> port;
}
value_set<vsk_t>(4) pvs;
```

This value_set's local name is `pvs`.

18.3.1.2. Tables For each `table` construct, its syntactic name becomes the local name of the table. For example:

```
control c(/* parameters omitted */)() {
    table t { /* body omitted */ }
```

```
}
```

This table's local name is `t`.

18.3.1.3. Keys

Syntactically, table keys are expressions. For simple expressions, the local key name can be generated from the expression itself; the algorithm by which a compiler derives control-plane names for complex key expressions is target-dependent.

The spec suggests, but does not mandate, the following algorithm for generating names for some kinds of key expressions:

Kind	Example	Name
The `isValid()` method.	`h.isValid()`	`"h.isValid()"`
Array accesses.	`header_stack[1]`	`"header_stack[1]"`
Constants.	`1`	`"1"`
Field projections.	`data.f1`	`"data.f1"`
Slices.	`f1[3:0]`	`"f1[3:0]"`
Masks.	`h.src & 0xFFFF`	`"h.src & 0xFFFF"`

In the following example, the previous algorithm would derive for table `t` two keys with names `data.f1` and `hdrs[3].f2`.

```
table t {
    keys = {
        data.f1 : exact;
        hdrs[3].f2 : exact;
    }
    actions = { /* body omitted */ }
}
```

If a compiler cannot generate a name for a key it requires the key expression to be annotated with a `@name` annotation (Section 20.3.3), as in the following example:

```
table t {
    keys = {
        data.f1 + 1 : exact @name("f1_mask");
    }
    actions = { /* body omitted */ }
}
```

Here, the `@name("f1_mask")` annotation assigns the local name `"f1_mask"` to this key.

18.3.1.4. Actions

For each `action` construct, its syntactic name is the local name of the action. For example:

```
control c(/* parameters omitted */)() {
    action a(...) { /* body omitted */ }
}
```

This action's local name is a.

18.3.1.5. Instances The local names of **extern**, **parser**, and **control** instances are derived based on how the instance is used. If the instance is bound to a name, that name becomes its local control plane name. For example, if **control** C is declared as,

```
control C(/* parameters omitted */)() { /* body omitted */ }
```

and instantiated as,

```
C() c_inst;
```

then the local name of the instance is c_inst.

Alternatively, if the instance is created as an actual argument, then its local name is the name of the formal parameter to which it will be bound. For example, if **extern** E and **control** C are declared as,

```
extern E { /* body omitted */ }
control C( /* parameters omitted */ )(E e_in) { /* body omitted */ }
```

and instantiated as,

```
C(E()) c_inst;
```

then the local name of the extern instance is e_in.

If the construct being instantiated is passed as an argument to a package, the instance name is derived from the user-supplied type definition when possible. In the following example, the local name of the instance of MyC is c, and the local name of the **extern** is e2, not e1.

```
extern E { /* body omitted */ }
control ArchC(E e1);
package Arch(ArchC c);

control MyC(E e2)() { /* body omitted */ }
Arch(MyC()) main;
```

Note that in this example, the architecture will supply an instance of the extern when it applies the instance of MyC passed to the Arch package. The fully-qualified name of that instance is main.c.e2.

Next, consider a larger example that demonstrates name generation when there are multiple instances.

```
control Callee() {
    table t { /* body omitted */ }
    apply { t.apply(); }
}
control Caller() {
    Callee() c1;
    Callee() c2;
```

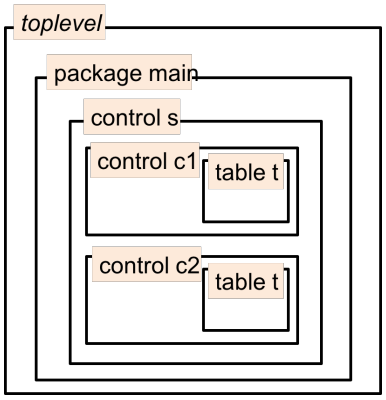

Figure 15. Evaluating a program that has several instantiations of the same component.

```
    apply {
       c1.apply();
       c2.apply();
    }
}
control Simple();
package Top(Simple s);
Top(Caller()) main;
```

The compile-time evaluation of this program produces the structure in Figure 15. Notice that there are two instances of the `table t`. These instances must both be exposed to the control plane. To name an object in this hierarchy, one uses a path composed of the names of containing instances. In this case, the two tables have names `s.c1.t` and `s.c2.t`, where `s` is the name of the argument to the package instantiation, which is derived from the name of its corresponding formal parameter.

18.3.2. Annotations controlling naming

Control plane-related annotations (Section 20.3.3) can alter the names exposed to the control plane in the following ways.

- The `@hidden` annotation hides a controllable entity from the control plane. This is the only case in which a controllable entity is not required to have a unique, fully-qualified name.
- The `@name` annotation may be used to change the local name of a controllable entity.

Programs that yield the same fully-qualified name for two different controllable entities are invalid.

18.3.3. Recommendations

The control plane may refer to a controllable entity by a postfix of its fully qualified name when it is unambiguous in the context in which it is used. Consider the following example.

```
control c( /* parameters omitted */ )() {
    action a ( /* parameters omitted */ ) { /* body omitted */ }
    table t {
        keys = { /* body omitted */ }
        actions = { a; } }
}
c() c_inst;
```

Control plane software may refer to action `c_inst.a` as `a` when inserting rules into table `c_inst.t`, because it is clear from the definition of the table which action `a` refers to.

Not all unambiguous postfix shortcuts are recommended. For instance, consider the first example in Section 18.3. One might be tempted to refer to `s.c1` simply as `c1`, as no other instance named `c1` appears in the program. However, this leads to a brittle program since future modifications can never introduce an instance named `c1`, or include libraries of P4 code that contain instances with that name.

18.4. Dynamic evaluation

The dynamic evaluation of a P4 program is orchestrated by the architecture model. Each architecture model needs to specify the order and the conditions under which the various P4 component programs are dynamically executed. For example, in the Simple Switch example from Section 5.1 the execution flow goes `Parser->Pipe->Deparser`.

Once a P4 execution block is invoked its execution proceeds until termination according to the semantics defined in this document.

18.4.1. Concurrency model

A typical packet processing system needs to execute multiple simultaneous logical "threads." At the very least there is a thread executing the control plane, which can modify the contents of the tables. Architecture specifications should describe in detail the interactions between the control-plane and the data-plane. The data plane can exchange information with the control plane through **`extern`** function and method calls. Moreover, high-throughput packet-processing systems may be processing multiple packets simultaneously, e.g., in a pipelined fashion, or concurrently parsing a first packet while performing match-action operations on a second packet. This section specifies the semantics of P4 programs with respect to such concurrent executions.

Each top-level **`parser`** or **`control`** block is executed as a separate thread when invoked by the architecture. All the parameters of the block and all local variables are thread-local—i.e., each thread has a private copy of these resources. This applies to the `packet_in` and `packet_out` parameters of parsers and deparsers.

As long as a P4 block uses only thread-local storage (e.g., metadata, packet headers, local variables), its behavior in the presence of concurrency is identical with the behavior in isolation, since any interleaving of statements from different threads must produce the same output.

In contrast, **`extern`** blocks instantiated by a P4 program are global, shared across all threads. If **`extern`** blocks mediate access to state (e.g., counters, registers)—i.e., the methods of the **`extern`** block read and write state, these stateful operations are subject to data races. P4 mandates that execution of a method call on an extern instance is atomic.

To allow users to express atomic execution of larger code blocks, P4 provides an `@atomic` annotation, which can be applied to block statements, parser states, control blocks, or whole parsers.

Consider the following example:

```
extern Register { /* body omitted */ }
control Ingress() {
  Register() r;
  table flowlet { /* read state of r in an action */ }
  table new_flowlet { /* write state of r in an action */ }
  apply {
    @atomic {
       flowlet.apply();
       if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TIMEOUT)
          new_flowlet.apply();
}}}
```

This program accesses an extern object `r` of type `Register` in actions invoked from tables `flowlet` (reading) and `new_flowlet` (writing). Without the `@atomic` annotation these two operations would not execute atomically: a second packet may read the state of `r` before the first packet had a chance to update it.

Note that even within an **`action`** definition, if the action does something like reading a register, modifying it, and writing it back, in a way that only the modified value should be visible to the next packet, then, to guarantee correct execution in all cases, that portion of the action definition should be enclosed within a block annotated with `@atomic`.

A compiler backend must reject a program containing `@atomic` blocks if it cannot implement the atomic execution of the instruction sequence. In such cases, the compiler should provide reasonable diagnostics.

19. Static assertions

The P4 core library contains two overloaded declarations for a `static_assert` function, as follows:

```
/// Static assert evaluates a boolean expression
/// at compilation time.  If the expression evaluates to
/// false, compilation is stopped and the corresponding message is printed.
extern bool static_assert(bool check, string message);

/// Like the above but using a default message.
extern bool static_assert(bool check);
```

These functions both return boolean values. Since the parameters are directionless, these functions require compile-time known values as arguments, thus they can be used to enforce compile-time invariants. Since P4 does not allow statements at the program top-level (outside of apply blocks), these functions can be used at the top-level by assigning their result to a dummy constant, e.g.:

```
const bool _check = static_assert(V1MODEL_VERSION > 20180000,
                                  "Expected a v1 model version >= 20180000");
```

As the comment indicates, if `static_assert` returns **`false`**, it causes the program compilation to be ter-

minated immediately with an error.

20. Annotations

Annotations are a simple mechanism for extending the P4 language to some limited degree without changing the grammar. Annotations are attached to types, fields, variables, etc. using the `@` syntax (as shown explicitly in the P4 grammar). Unstructured annotations, or just "annotations," have an optional body; structured annotations have a mandatory body, containing at least a pair of square brackets `[]`.

```
optAnnotations
    : /* empty */
    | annotations
    ;

annotations
    : annotation
    | annotations annotation
    ;

annotation
    : "@" name
    | "@" name "(" annotationBody ")"
    | "@" name "[" structuredAnnotationBody "]"
    ;
```

Structured annotations and unstructured annotations on any one element must not use the same `name`. Thus, a given `name` can only be applied to one type of annotation or the other for any one element. An annotation used on one element does not affect the annotation on another because they have different scope.

This is legal:

```
@my_anno(1) table T { /* body omitted */ }
@my_anno[2] table U { /* body omitted */ } // OK - different scope than previous
                                           // use of my_anno
```

This is illegal:

```
@my_anno(1)
@my_anno[2] table U { /* body omitted */ } // Error - changed type of anno
                                           // on an element
```

Multiple unstructured annotations using the same `name` can appear on a given element; they are cumulative. Each one will be bound to that element. In contrast, only one structured annotation using a given `name` may appear on an element; multiple uses of the same `name` will produce an error.

This is legal:

```
@my_anno(1)
@my_anno(2) table U { /* body omitted */ }  // OK - unstructured annos accumulate
```

This is illegal:

```
@my_anno[1]
@my_anno[2] table U { /* body omitted */ } // Error - reused the same structured
                                           // anno on an element
```

20.1. Bodies of Unstructured Annotations

The flexibility of P4 unstructured annotations comes from the minimal structure mandated by the P4 grammar: unstructured annotation bodies may contain any sequence of terminals, so long as parentheses are balanced. In the following grammar fragment, the `annotationToken` non-terminal represents any terminal produced by the lexer, including keywords, identifiers, string and integer literals, and symbols, but excluding parentheses.

```
annotationBody
    : /* empty */
    | annotationBody "(" annotationBody ")"
    | annotationBody annotationToken
    ;
```

Unstructured annotations may impose additional structure on their bodies, and are not confined to the P4 language. For example, the P4Runtime specification[4] defines a `@pkginfo` annotation that expects key-value pairs.

20.2. Bodies of Structured Annotations

Unlike unstructured annotations, structured annotations use square brackets `[...]` and have a restricted format. They are commonly used to declare custom metadata, consisting of expression lists or key-value lists but not both. An `expressionList` may be empty or contain a comma-separated list of member `expressions`. A `kvList` consists of one or more `kvPairs`, each consisting of a key and a value `expression`. Note the syntax for `expression` is rich, see Appendix G for details.

All `expressions` within a `structuredAnnotationBody` must be compile-time known values with a result type that is either: **`string`**, **`int`**, or **`bool`**. In particular, structured `expressions` (e.g. an `expression` containing an `expressionList`, a `kvList`, etc.) are not allowed. Note that P4Runtime information (P4Info) may stipulate additional restrictions. For example, an integer expression might be limited to 64-bit values.

It is illegal to duplicate a `key` within the `kvList` of a structured annotation.

```
structuredAnnotationBody
    : expressionList optTrailingComma
    | kvList optTrailingComma
```

[4] The P4Runtime API is defined as a Google Protocol Buffer `.proto` file and an accompanying English specification document here: https://github.com/p4lang/p4runtime

```
    ;
...
expressionList
    : /* empty */
    | expression
    | expressionList "," expression
    ;
...
kvList
    : kvPair
    | kvList "," kvPair
    ;

kvPair
    : name "=" expression
    ;
```

20.2.1. Structured Annotation Examples

Empty Expression List

The following example produces an empty annotation:

```
@Empty[]
table t {
    /* body omitted */
}
```

Mixed Expression List

The following example will produce an effective expression list as follows:

```
[1,"hello",true, false, 11]
```

```
#define TEXT_CONST "hello"
#define NUM_CONST 6
@MixedExprList[1,TEXT_CONST,true,1==2,5+NUM_CONST]
table t {
    /* body omitted */
}
```

kvList of Strings

```
@Labels[short="Short Label", hover="My Longer Table Label to appear in hover-help"]
table t {
    /* body omitted */
}
```

kvList of Mixed Expressions

The following example will produce an effective kvList as follows.

```
[label="text", my_bool=true, int_val=6]
```

```
@MixedKV[label="text", my_bool=true, int_val=2*3]
table t {
    /* body omitted */
}
```

Illegal Mixing of **kvPair** and **expressionList**

The following example is invalid because the body contains both a `kvPair` and an `expression`:

```
@IllegalMixing[key=4, 5] // illegal mixing
table t {
    /* body omitted */
}
```

Illegal Duplicate Key

The following example is invalid because the same key occurs more than once:

```
@DupKey[k1=4,k1=5] // illegal duplicate key
table t {
    /* body omitted */
}
```

Illegal Duplicate Structured Annotation

The following example is invalid because the annotation `name` occurs more than once on the same element, e.g. **table** `t`:

```
@DupAnno[k1=4]
@DupAnno[k2=5] // illegal duplicate name
table t {
    /* body omitted */
}
```

Illegal Simultaneous Use of Both Structured and Unstructured Annotation

The following example is invalid because the annotation `name` is used by both an unstructured and structured annotation on the same element **table** `t`:

```
@MixAnno("Anything")
@MixAnno[k2=5] // illegal use in both annotation types
table t {
    /* body omitted */
}
```

20.3. Predefined annotations

Annotation names that start with lowercase letters are reserved for the standard library and architecture. This document pre-defines a set of "standard" annotations in Appendix C. We expect that this list will grow. We encourage custom architectures to define annotations starting with a manufacturer prefix: e.g., an organization named X would use annotations named like `@X_annotation`

20.3.1. Optional parameter annotations

A parameter to a package, parser type, control type, extern method, extern function or extern object constructor can be annotated with `@optional` to indicate that the user does not need to provide a corresponding argument for that parameter. The meaning of a parameter with no supplied value is target-dependent.

20.3.2. Annotations on the table action list

The following two annotations can be used to give additional information to the compiler and control-plane about actions in a table. These annotations have no bodies.

- `@tableonly`: actions with this annotation can only appear within the table, and never as default action.
- `@defaultonly`: actions with this annotation can only appear in the default action, and never in the table.

```
table t {
    actions = {
       a,              // can appear anywhere
       @tableonly b,   // can only appear in the table
       @defaultonly c, // can only appear in the default action
    }
    /* body omitted */
}
```

20.3.3. Control-plane API annotations

The `@name` annotation directs the compiler to use a different local name when generating the external APIs used to manipulate a language element from the control plane. This annotation takes a string literal body. In the following example, the fully-qualified name of the table is `c_inst.t1`.

```
control c( /* parameters omitted */ )() {
    @name("t1") table t { /* body omitted */ }
    apply { /* body omitted */ }
}
c() c_inst;
```

The `@hidden` annotation hides a controllable entity, e.g. a table, key, action, or extern, from the control plane. This effectively removes its fully-qualified name (Section 18.3). This annotation does not have a body.

20.3.3.1. Restrictions Each element may be annotated with at most one `@name` or `@hidden` annotation, and each control plane name must refer to at most one controllable entity. This is of special concern when using an absolute `@name` annotation: if a type containing a `@name` annotation with an absolute pathname (i.e., one starting with a dot) is instantiated more than once, it will result in the same name referring to two controllable entities.

```
control noargs();
package top(noargs c1, noargs c2);

control c() {
    @name(".foo.bar") table t { /* body omitted */ }
    apply { /* body omitted */ }
}
top(c(), c()) main;
```

Without the `@name` annotation, this program would produce two controllable entities with fully-qualified names `main.c1.t` and `main.c2.t`. However, the `@name(".foo.bar")` annotation renames table `t` in both instances to `foo.bar`, resulting in one name that refers to two controllable entities, which is illegal.

20.3.4. Concurrency control annotations

The `@atomic` annotation, described in Section 18.4.1 can be used to enforce the atomic execution of a code block.

20.3.5. Value set annotations

The `@match` annotation, described in Section 13.6, is used to specify a **`match_kind`** value other than the default **`match_kind`** of `exact` for a field of a **`value_set`**.

20.3.6. Extern function/method annotations

Various annotations may appear on extern function and method declarations to describe limitations on the behavior and interactions of those functions. By default extern functions might have any effect on the environment of the P4 program and might interact in non-trivial ways (subject to a few limitations – see section 6.8.1). Since externs are architecture-specific and their behavior is known to the architecture definition, these annotations are not strictly necessary (an implementation can have knowledge of how externs interact based on their names built into it), but these annotations provide a uniform way of describing certain well-defined interactions (or their absence), allowing architecture-independent analysis of P4 programs.

- `@pure` - Describes a function that depends solely on its **`in`** parameter values, and has no effect other than returning a value, and copy-out behavior on its **`out`** and **`inout`** parameters. No hidden state is recorded between calls, and its value does not depend on any hidden state that may be changed by other calls. An example is a `hash` function that computes a deterministic hash of its arguments, and its return value does not depend upon any control-plane writable seed or initialization vector value. A `@pure` function whose results are unused may be safely eliminated with no adverse effects, and multiple calls with identical arguments may be combined into a single call

(subject to the limits imposed by copy-out behavior of `out` and `inout` parameters). `@pure` functions may also be reordered with respect to other computations that are not data dependent.

- `@noSideEffects` - Weaker than `@pure` and describes a function that does not change any hidden state, but may depend on hidden state. One example is a `hash` function that computes a deterministic hash of its arguments, plus some internal state that can be modified via control plane API calls such as a seed or initialization vector. Another example is a read of one element of a register array extern object. Such a function may be dead code eliminated, and may be reordered or combined with other `@noSideEffects` or `@pure` calls (subject to the limits imposed by copy-out behavior of `out` and `inout` parameters), but not with other function calls that may have side effects that affect the function.

20.3.7. Deprecated annotation

The `deprecated` annotation has a required string argument that is a message that will be printed by a compiler when a program is using the deprecated construct. This is mostly useful for annotating library constructs, such as externs.

```
@deprecated("Please use the 'check' function instead")
extern Checker {
   /* body omitted */
}
```

20.3.8. No warnings annotation

The `noWarn` annotation has a required string argument that indicates a compiler warning that will be inhibited. For example `@noWarn("unused")` on a declaration will prevent a compiler warning if that declaration is not used.

20.4. Target-specific annotations

Each P4 compiler implementation can define additional annotations specific to the target of the compiler. The syntax of the annotations should conform to the above description. The semantics of such annotations is target-specific. They could be used in a similar way to pragmas in other languages.

The P4 compiler should provide:

- Errors when annotations are used incorrectly (e.g., an annotation expecting a parameter but used without arguments, or with arguments of the wrong type
- Warnings for unknown annotations.

A. Appendix: Revision History

A.1. Summary of changes made in version 1.2.4

- Introduced distinction between local compile-time known and compile-time known values (Section18.1).
- Added header stack expressions (Section 8.18.1).

- Allow casts from a type to itself (Section 8.11).
- Added an invalid header or header union expression `{#}` (Sections 8.17 and 8.19).
- Added a concept of numeric values (Section 7.4).
- Added a section on operations on extern objects (Section 8.22).
- Added note in sections operations on types for types that support compile-time size determination.
- Clarified that header stacks are arrays of headers or header unions.
- Added distinctness of fields for types that have fields including error, match kind, struct, header, and header union.
- Clarified types **`bit`**`<W>`, **`int`**`<W>`, and **`varbit`**`<W>` encompass the case where the width is a compile-time known expression evaluating to an appropriate integer (Section 7.1.6.2, Section 7.1.6.3, Section 7.1.6.4).
- Clarified restrictions for parameters with default values (Section 6.8.1).
- Added optional trailing commas (Section 6.4.4).
- Clarified the scope of parser namespaces (Section 13.2).
- Specified that algorithm for generating control-plane names for keys is optional (Section 18.3.1.3).
- Clarified types of expressions that may appear in **`select`** (Section 13.6).
- Added description of semantics of the core.p4 match kinds (Section 14.2.1.1).
- Explicitly disallow overloading of parsers, controls, and packages (Section 7.2.10.2).
- Clarified implicit casts present in select expressions (Section 13.6).
- Clarified that slices can be applied to arbitrary-precision integers (Section 8.8).
- Clarified that direct invocation is not possible for objects that have constructor arguments (Section 15.1).
- Added comparison for tuples as a legal operation (Section 8.12).
- Clarified the behavior of `lookahead` on header-typed values (Section 13.8.3).
- Added `static_assert` function (Section 19).
- Clarified semantics of ranges where the start is bigger than the end (Section 8.15.4).
- Allow ranges to be specified by serializable enums (Section 8.15.4).
- Specified type produced by the `*sizeInB*` methods (Section 9).
- Added section with operations on **`match_kind`** values (Section 8.4).
- Renamed infinite-precision integers to arbitrary-precision integers (Section 7.1.6.5).
- compiler-inserted `default_action` is not **`const`** (Section 14.2).
- Clarified the restrictions on run time for tables with **`const entries`** (Section 14.2.1.4).
- renamed list expressions to tuple expressions
- Added **`list`** type (Section 7.2.7).
- Defined **`entries`** table property without **`const`**, for entries installed when the P4 program is loaded, but the control plane can later change them or add to them (Section 14.2.1.4).
- Clarified behavior of table with no `key` property, or if its list of keys is empty (Section 14.2.1.1).

A.2. Summary of changes made in version 1.2.3, released July 11, 2022.

- Extended `minSizeInBits` and `minSizeInBytes` to apply to more expressions (Section 9).
- Added support for `maxSizeInBits` and `maxSizeInBytes` (Section 9).
- Added support for empty lists of const entries in tables (Section 14.2.1.4).
- Added support for **`switch`** statements in actions (Section 14.1).
- Added support for direct invocation of controls and parsers (Section 15).

- Added parser **value_set** to list of control-plane visible names (Section 18.3).
- Added **match_kind** as a base type (Section 7.1.3).
- Removed structure initializers as they are subsumed by structure-valued expressions (Section 8.13).
- Specified operations on values typed as type variables (Section 8.24).
- Clarified semantics of compile-time known values (Section 18.1).
- Clarified semantics of directionless parameters (Section 6.8).
- Clarified semantics of arbitrary precision integers (Section 7.1.6.5).
- Clarified semantics of bit slices, shifts, and concatenation (Section 8.6).
- Clarified semantics of optional parameters (Section 6.8.2).
- Clarified restrictions on extern method and function invocation (Section F).
- Clarified semantics of implicit casts (Section 8.11.2).

A.3. Summary of changes made in version 1.2.2, released May 17, 2021

- Added support for accessing tuple fields (Section 8.12).
- Added support for generic structures (Section 7.2.11).
- Added support for integers, **enum**s, and **error**s in **switch** statements (Section 12.7).
- Added support for additional enumeration types (Section 7.2.1).
- Added support for abstract methods (Section 1).
- Added support for conditional statements and empty statements in parsers (Section 13.4).
- Added support for casts from **int** to **bool** (Section 8.11).
- Added support for 0-width bitstrings and varbits (Section 8.25).
- Clarified that `default_action` is `NoAction` if otherwise unspecified (Section 14.2).
- Clarified the types of expressions that may be used as indexes for header stacks (Section 8.18).
- Clarified representation of Booleans in headers (Section 7.2.2).
- Clarified representation of empty types (Section 8.25).
- Clarified that action data can be specified by the control plane, `default_action` table property, or **const entries** table property (Section 14.1).
- Fixed several typos and inconsistencies in grammar (Section G).
- Eliminated annotations on **const** entries in grammar (Section G).

A.4. Summary of changes made in version 1.2.1, released June 11, 2020

- Added structure-value expressions (Section 8.13).
- Added support for default values (Section 7.3).
- Added support for concatenating signed strings (Section 8.9.1).
- Added key-value and list-structured annotations (Section 20).
- Added `@pure` and `@noSideEffects` annotations (Section 20.3.6).
- Added `@noWarn` annotation (Section 20.3.8).
- Generalized typing for masks to allow serializable **enum**s (Section 8.15.3).
- Restricted the right operands of bit shifts involving arbitrary-precision integers to be constant and positive (Section 8.8).
- Clarified copy-out behavior for **return** (Section 12.4) and **exit** (Section 12.5) statements.
- Clarified semantics of invalid header stacks (Section 8.25).
- Clarified initialization semantics (Section 6.7 and 6.8), especially for headers and local variables.
- Clarified evaluation order for table keys (Section 14.2.3).

- Fixed grammar to clarify parsing of right shift operator (>>), allow empty statements in parser (Section 13.4), and eliminate annotations on const entries (Section 14.2.1.4).

A.5. Summary of changes made in version 1.2.0, released October 14, 2019

- Added `table.apply().miss` (Section 14.2.2).
- Added `string` type (Section 7.1.5).
- Added implicit casts from enum values (Section 8.3).
- Allow 1-bit signed values
- Define the type of bit slices from signed and unsigned values to be unsigned.
- Constrain `default` label position for `switch` statements.
- Allow empty tuples.
- Added `@deprecated` annotation.
- Relaxed the structure of annotation bodies.
- Removed the `@pkginfo` annotation, which is now defined by the P4Runtime specification.
- Added `int` type (Section 7.1.6.5).
- Added error `ParserInvalidArgument` (Sections 13.8.2, 13.8.4).
- Clarified the significance of order of entries in `const entries` (Section 14.2.1.4).
- Added methods to calculate header size (Section 8.17).

A.6. Summary of changes made in version 1.1.0, released November 26, 2017.

- Top-level functions (Section 10)
  - Functions may be declared at the top-level of a P4 program.
- Optional and named parameters (Section 6.8)
  - Parameters may be specified by name, with a default value, or designated as optional.
- `enum` representations (Section 8.3)
  - `enum` values to be specified with a concrete representation.
- Parser values sets (Section 13.11)
  - `value_set` objects for control-plane programmable `select` labels.
- Type definitions (Section 7.6)
  - New types may be introduced in programs.
- Saturating arithmetic (Section 8.6)
  - Saturating arithmetic is supported on some targets.
- Structured annotations (Section 20)
  - Annotations may be specified as lists of key-value pairs
- Globalname (Section 18.3.2)
  - The reserved `globalname` annotation has been removed.

- Table `size` property (Section 14.2.1.5)
  - Meaning of optional `size` property for tables has been defined.
- Invalid headers (Section 8.17)
  - Clarified semantics of operations on invalid headers.
- Calling restrictions (Section F)
  - Added restrictions on kinds of values that may be passed as arguments to calls.
- Bitwise operator precedence (Section G)
  - Modified precedence conventions so that bitwise operators `&` `|` and `^` have higher precedence than relation operators `<` `>` `<=` `>=`.
- Computed bitwidths (Section 7.1)
  - Added support for specifying widths using expressions in **`bit`** and **`varbit`** types.

A.7. Initial version 1.0.0, released May 17, 2017

B. Appendix: P4 reserved keywords

The following table shows all P4 reserved keywords. Some identifiers are treated as keywords only in specific contexts (e.g., the keyword `actions`).

`abstract`	**`action`**	**`apply`**	**`bit`**
`bool`	**`const`**	**`control`**	**`default`**
`else`	**`enum`**	**`error`**	**`extern`**
`exit`	**`false`**	**`header`**	**`header_union`**
`if`	**`in`**	**`inout`**	**`int`**
`list`	**`match_kind`**	**`package`**	**`parser`**
`out`	**`return`**	**`select`**	**`state`**
`string`	**`struct`**	**`switch`**	**`table`**
`this`	**`transition`**	**`true`**	**`tuple`**
`type`	**`typedef`**	**`value_set`**	**`varbit`**
`verify`	**`void`**		

C. Appendix: P4 reserved annotations

The following table shows all P4 reserved annotations.

Annotation	Purpose	See Section
`atomic`	specify atomic execution	18.4.1
`defaultonly`	action can only appear in the default action	20.3.2
`hidden`	hides a controllable entity from the control plane	18.3.2

match	specify match_kind of a field in a value_set	20.3.5
name	assign local control-plane name	18.3.2
optional	parameter is optional	20.3.1
tableonly	action cannot be a default_action	20.3.2
deprecated	Construct has been deprecated	20.3.7
pure	pure function	20.3.6
noSideEffects	function with no side effects	20.3.6
noWarn	Has a string argument; inhibits compiler warnings	20.3.8

D. Appendix: P4 core library

The P4 core library contains declarations that are useful to most programs.

For example, the core library includes the declarations of the predefined `packet_in` and `packet_out` extern objects, used in parsers and deparsers to access packet data.

```
/// Standard error codes.  New error codes can be declared by users.
error {
    NoError,           /// No error.
    PacketTooShort,    /// Not enough bits in packet for 'extract'.
    NoMatch,           /// 'select' expression has no matches.
    StackOutOfBounds,  /// Reference to invalid element of a header stack.
    HeaderTooShort,    /// Extracting too many bits into a varbit field.
    ParserTimeout,     /// Parser execution time limit exceeded.
    ParserInvalidArgument  /// Parser operation was called with a value
                           /// not supported by the implementation.
}
extern packet_in {
    /// Read a header from the packet into a fixed-sized header @hdr
    /// and advance the cursor.
    /// May trigger error PacketTooShort or StackOutOfBounds.
    /// @T must be a fixed-size header type
    void extract<T>(out T hdr);
    /// Read bits from the packet into a variable-sized header @variableSizeHeader
    /// and advance the cursor.
    /// @T must be a header containing exactly 1 varbit field.
    /// May trigger errors PacketTooShort, StackOutOfBounds, or HeaderTooShort.
    void extract<T>(out T variableSizeHeader,
                    in bit<32> variableFieldSizeInBits);
    /// Read bits from the packet without advancing the cursor.
    /// @returns: the bits read from the packet.
    /// T may be an arbitrary fixed-size type.
    T lookahead<T>();
    /// Advance the packet cursor by the specified number of bits.
    void advance(in bit<32> sizeInBits);
    /// @return packet length in bytes.  This method may be unavailable on
```

```
    /// some target architectures.
    bit<32> length();
}
extern packet_out {
    /// Write @data into the output packet, skipping invalid headers
    /// and advancing the cursor
    /// @T can be a header type, a header stack, a header_union, or a struct
    /// containing fields with such types.
    void emit<T>(in T data);
}
action NoAction() {}
/// Standard match kinds for table key fields.
/// Some architectures may not support all these match kinds.
/// Architectures can declare additional match kinds.
match_kind {
    /// Match bits exactly.
    exact,
    /// Ternary match, using a mask.
    ternary,
    /// Longest-prefix match.
    lpm
}

/// Static assert evaluates a boolean expression
/// at compilation time.  If the expression evaluates to
/// false, compilation is stopped and the corresponding message is printed.
/// The function returns a boolean, so that it can be used
/// as a global constant value in a program, e.g.:
/// const version = static_assert(V1MODEL_VERSION > 20180000, "Expected a v1 model version >= 20180000");
extern bool static_assert(bool check, string message);

/// Like the above but using a default message.
extern bool static_assert(bool check);
```

E. Appendix: Checksums

There are no built-in constructs in $P4_{16}$ for manipulating packet checksums. We expect that checksum operations can be expressed as **extern** library objects that are provided in target-specific libraries. The standard architecture library should provide such checksum units.

For example, one could provide an incremental checksum unit `Checksum16` (also described in the VSS example in Section 5.2.4) for computing 16-bit one's complement using an **extern** object with a signature such as:

```
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

IP checksum verification could be done in a parser as:

```
ck16.clear();           // prepare checksum unit
ck16.update(h.ipv4);    // write header
verify(ck16.get() == 16w0, error.IPv4ChecksumError); // check for 0 checksum
```

IP checksum generation could be done as:

```
h.ipv4.hdrChecksum = 16w0;
ck16.clear();
ck16.update(h.ipv4);
h.ipv4.hdrChecksum = ck16.get();
```

Moreover, some switch architectures do not perform checksum verification, but only update checksums incrementally to reflect packet modifications. This could be achieved as well, as the following P4 program fragments illustrates:

```
ck16.clear();
ck16.update(h.ipv4.hdrChecksum);  // original checksum
ck16.remove( { h.ipv4.ttl, h.ipv4.proto } );
h.ipv4.ttl = h.ipv4.ttl - 1;
ck16.update( { h.ipv4.ttl, h.ipv4.proto } );
h.ipv4.hdrChecksum = ck16.get();
```

F. Appendix: Restrictions on compile time and run time calls

This appendix summarizes restrictions on compile time and run time calls that can be made. Many of them are described earlier in this document, but are collected here for easy reference.

The stateful types of objects in $P4_{16}$ are packages, parsers, controls, externs, tables, and value-sets. $P4_{16}$ functions are also considered to be in that group, even if they happen to be pure functions of their arguments. All other types are referred to as “value types” here.

Some guiding principles:

- Controls are not allowed to call parsers, and vice versa, so there is no use in passing one type to the other in constructor parameters or run-time parameters.
- At run time, after a control is called, and before that call is complete, there can be no recursive calls between controls, nor from a control to itself. Similarly for parsers. There can be loops

among states within a single parser.

- Externs are not allowed to call parsers or controls, so there is no use in passing objects of those types to them.
- Tables are always instantiated directly in their enclosing control, and cannot be instantiated at the top level. There is no syntax for specifying parameters that are tables. Tables are only intended to be used from within the control where they are defined.
- Value-sets can be instantiated in an enclosing parser or at the top level. There is no syntax for specifying parameters that are value-sets. Value-sets can be shared between the parsers as long as they are in the scope.

A note on recursion: It is expected that some architectures will define capabilities for recirculating a packet to be processed again as if it were a newly arriving packet, or to make "clones" of packets that are then processed by parsers and/or control blocks that the original packet has already completed. This does not change the notes above on recursion that apply while a parser or control is executing.

The first table lists restrictions on what types can be passed as constructor parameters to other types.

	can be a constructor parameter for this type			
This type	package	parser	control	extern
package	yes	no	no	no
parser	yes	yes	no	no
control	yes	no	yes	no
extern	yes	yes	yes	yes
function	no	no	no	no
table	no	no	no	no
value-set	no	no	no	no
value types	yes	yes	yes	yes

The next table lists restrictions on where one may perform instantiations (see Section 11.3) of different types. The answer for **package** is always "no" because there is no "inside a package" where instantiations can be written in $P4_{16}$. One can definitely make constructor calls and use instances of stateful types as parameters when instantiating a package, and restrictions on those types are in the table above.

For externs, one can only specify their interface in $P4_{16}$, not their implementation. Thus there is no place to instantiate objects within an extern.

You may declare variables and constants of any of the value types within a parser, control, or function (see Section 11.2 for more details). Declaring a variable or constant is not the same as instantiation, hence the answer "N/A" (for not applicable) in those table entries. Variables may not be declared at the top level of your program, but constants may.

	can be instantiated in this place					
This type	top level	package	parser	control	extern	function
package	yes	no	no	no	no	no
parser	no	no	yes	no	no	no
control	no	no	no	yes	no	no
extern	yes	no	yes	yes	no	no
function	yes	no	no	no	no	no

table	no	no	no	yes	no	no
value-set	yes	no	yes	no	no	no
value types	N/A	N/A	N/A	N/A	N/A	N/A

The next table lists restrictions on what types can be passed as run-time parameters to other callable things that have run-time parameters: parsers, controls, externs (including methods and extern functions), actions, and functions.

	can be a run-time parameter to this callable thing				
This type	parser	control	extern	action	function
package	no	no	no	no	no
parser	no	no	no	no	no
control	no	no	no	no	no
extern	yes	yes	yes	no	no
table	no	no	no	no	no
value-set	no	no	no	no	no
action	no	no	no	no	no
function	no	no	no	no	no
value types	yes	yes	yes	yes	yes

Extern method and extern function calls may only return a value that is a value type, or no value at all (specified by a return type of **void**).

The next table lists restrictions on what kinds of calls can be made from which places in a P4 program. Calling a parser, control, or table means invoking its **apply()** method. Calling a value-set means using it in a select expression. The row for **extern** describes where extern method calls can be made from.

One way that an extern can be called from the top level of a parser or control is in an initializer expression for a declared variable, e.g. `bit<32> x = rand.get();`.

	can be called at run time from this place in a P4 program					
This type	parser state	control apply block	parser or control top level	action	extern	function
package	N/A	N/A	N/A	N/A	N/A	N/A
parser	yes	no	no	no	no	no
control	no	yes	no	no	no	no
extern	yes	yes	yes	yes	no	no
table	no	yes	no	no	no	no
value-set	yes	no	no	no	no	no
action	no	yes	no	yes	no	no
function	yes	yes	no	yes	no	yes
value types	N/A	N/A	N/A	N/A	N/A	N/A

There may not be any recursion in calls, neither by a thing calling itself directly, nor mutual recursion.

An extern can never cause any other type of P4 program object to be called. See Section 6.8.1.

Actions may be called directly from a control **apply** block.

Note that while the extern row shows that extern methods can be called from many places, partic-

ular externs may have additional restrictions not listed in this table. Any such restrictions should be documented in the description for each extern, as part of the documentation for the architecture that defines the extern.

In many cases, the restriction will be "from a parser state only" or "from a control apply block or action only", but it may be even more restrictive, e.g. only from a particular kind of control block instantiated in a particular role in an architecture.

G. Appendix: P4 grammar

This is the grammar of $P4_{16}$ written using the YACC/bison language. Absent from this grammar is the precedence of various operations.

The grammar is actually ambiguous, so the lexer and the parser must collaborate for parsing the language. In particular, the lexer must be able to distinguish two kinds of identifiers:

- Type names previously introduced (`TYPE_IDENTIFIER` tokens)
- Regular identifiers (`IDENTIFIER` token)

The parser has to use a symbol table to indicate to the lexer how to parse subsequent appearances of identifiers. For example, given the following program fragment:

```
typedef bit<4> t;
struct s { /* body omitted */}
t x;
parser p(bit<8> b) { /* body omitted */ }
```

The lexer has to return the following terminal kinds:

```
t - TYPE_IDENTIFIER
s - TYPE_IDENTIFIER
x - IDENTIFIER
p - TYPE_IDENTIFIER
b - IDENTIFIER
```

This grammar has been heavily influenced by limitations of the Bison parser generator tool.

The `STRING_LITERAL` token corresponds to a string literal enclosed within double quotes, as described in Section 6.4.3.3.

All other terminals are uppercase spellings of the corresponding keywords. For example, `RETURN` is the terminal returned by the lexer when parsing the keyword return.

```
p4program
    : /* empty */
    | p4program declaration
    | p4program ";"  /* empty declaration */
    ;

declaration
    : constantDeclaration
```

```
    | externDeclaration
    | actionDeclaration
    | parserDeclaration
    | typeDeclaration
    | controlDeclaration
    | instantiation
    | errorDeclaration
    | matchKindDeclaration
    | functionDeclaration
    ;

nonTypeName
    : IDENTIFIER
    | APPLY
    | KEY
    | ACTIONS
    | STATE
    | ENTRIES
    | TYPE
    | PRIORITY
    ;

name
    : nonTypeName
    | LIST
    | TYPE_IDENTIFIER
    ;

nonTableKwName
   : IDENTIFIER
   | TYPE_IDENTIFIER
   | APPLY
   | STATE
   | TYPE
   | PRIORITY
   ;

optCONST
    : /* empty */
    | CONST
    ;

optAnnotations
    : /* empty */
    | annotations
    ;
```

```
annotations
    : annotation
    | annotations annotation
    ;

annotation
    : "@" name
    | "@" name "(" annotationBody ")"
    | "@" name "[" structuredAnnotationBody "]"
    ;

annotationBody
    : /* empty */
    | annotationBody "(" annotationBody ")"
    | annotationBody annotationToken
    ;

annotationToken
    : UNEXPECTED_TOKEN
    | ABSTRACT
    | ACTION
    | ACTIONS
    | APPLY
    | BOOL
    | BIT
    | CONST
    | CONTROL
    | DEFAULT
    | ELSE
    | ENTRIES
    | ENUM
    | ERROR
    | EXIT
    | EXTERN
    | FALSE
    | HEADER
    | HEADER_UNION
    | IF
    | IN
    | INOUT
    | INT
    | KEY
    | MATCH_KIND
    | TYPE
    | OUT
```

```
    | PARSER
    | PACKAGE
    | PRAGMA
    | RETURN
    | SELECT
    | STATE
    | STRING
    | STRUCT
    | SWITCH
    | TABLE
    | THIS
    | TRANSITION
    | TRUE
    | TUPLE
    | TYPEDEF
    | VARBIT
    | VALUESET
    | LIST
    | VOID
    | "_"
    | IDENTIFIER
    | TYPE_IDENTIFIER
    | STRING_LITERAL
    | INTEGER
    | "&&&"
    | ".."
    | "<<"
    | "&&"
    | "||"
    | "=="
    | "!="
    | ">="
    | "<="
    | "++"
    | "+"
    | "|+|"
    | "-"
    | "|-|"
    | "*"
    | "/"
    | "%"
    | "|"
    | "&"
    | "^"
    | "~"
    | "["
```

```
    | "]"
    | "{"
    | "}"
    | "<"
    | ">"
    | "!"
    | ":"
    | ","
    | "?"
    | "."
    | "="
    | ";"
    | "@"
    ;

kvList
    : kvPair
    | kvList "," kvPair
    ;

kvPair
    : name "=" expression
    ;

parameterList
    : /* empty */
    | nonEmptyParameterList
    ;

nonEmptyParameterList
    : parameter
    | nonEmptyParameterList "," parameter
    ;

parameter
    : optAnnotations direction typeRef name
    | optAnnotations direction typeRef name "=" expression
    ;

direction
    : IN
    | OUT
    | INOUT
    | /* empty */
    ;
```

```
packageTypeDeclaration
    : optAnnotations PACKAGE name optTypeParameters
      "(" parameterList ")"
    ;

instantiation
    : annotations typeRef "(" argumentList ")" name ";"
    | typeRef "(" argumentList ")" name ";"
    | annotations typeRef "(" argumentList ")" name "=" objInitializer ";"
    | typeRef "(" argumentList ")" name "=" objInitializer ";"
    ;

objInitializer
    : "{" objDeclarations "}"
    ;

objDeclarations
    : /* empty */
    | objDeclarations objDeclaration
    ;

objDeclaration
    : functionDeclaration
    | instantiation
    ;

optConstructorParameters
    : /* empty */
    | "(" parameterList ")"
    ;

dotPrefix
    : "."
    ;

/************************** PARSER *****************************/

parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      "{" parserLocalElements parserStates "}"
    ;

parserLocalElements
    : /* empty */
    | parserLocalElements parserLocalElement
    ;
```

```
parserLocalElement
    : constantDeclaration
    | instantiation
    | variableDeclaration
    | valueSetDeclaration
    ;

parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters
      "(" parameterList ")"
    ;

parserStates
    : parserState
    | parserStates parserState
    ;

parserState
    : optAnnotations STATE name
      "{" parserStatements transitionStatement "}"
    ;

parserStatements
    : /* empty */
    | parserStatements parserStatement
    ;

parserStatement
    : assignmentOrMethodCallStatement
    | directApplication
    | emptyStatement
    | variableDeclaration
    | constantDeclaration
    | parserBlockStatement
    | conditionalStatement
    ;

parserBlockStatement
    : optAnnotations "{" parserStatements "}"
    ;

transitionStatement
    : /* empty */
    | TRANSITION stateExpression
    ;
```

```
stateExpression
    : name ";"
    | selectExpression
    ;

selectExpression
    : SELECT "(" expressionList ")" "{" selectCaseList "}"
    ;

selectCaseList
    : /* empty */
    | selectCaseList selectCase
    ;

selectCase
    : keysetExpression ":" name ";"
    ;

keysetExpression
    : tupleKeysetExpression
    | simpleKeysetExpression
    ;

tupleKeysetExpression
    : "(" simpleKeysetExpression "," simpleExpressionList ")"
    | "(" reducedSimpleKeysetExpression ")"
    ;

optTrailingComma
    : /* empty */
    | ","
    ;

simpleExpressionList
    : simpleKeysetExpression
    | simpleExpressionList "," simpleKeysetExpression
    ;

reducedSimpleKeysetExpression
    : expression "&&&" expression
    | expression ".." expression
    | DEFAULT
    | "_"
    ;
```

```
simpleKeysetExpression
    : expression
    | expression "&&&" expression
    | expression ".." expression
    | DEFAULT
    | "_"
    ;

valueSetDeclaration
  : optAnnotations
      VALUESET "<" baseType ">" "(" expression ")" name ";"
  | optAnnotations
      VALUESET "<" tupleType ">" "(" expression ")" name ";"
  | optAnnotations
      VALUESET "<" typeName ">" "(" expression ")" name ";"
  ;

/************************** CONTROL ************************/

controlDeclaration
    : controlTypeDeclaration optConstructorParameters
      /* controlTypeDeclaration cannot contain type parameters */
      "{" controlLocalDeclarations APPLY controlBody "}"
    ;

controlTypeDeclaration
    : optAnnotations CONTROL name optTypeParameters
      "(" parameterList ")"
    ;

controlLocalDeclarations
    : /* empty */
    | controlLocalDeclarations controlLocalDeclaration
    ;

controlLocalDeclaration
    : constantDeclaration
    | actionDeclaration
    | tableDeclaration
    | instantiation
    | variableDeclaration
    ;

controlBody
    : blockStatement
    ;
```

```
/************************** EXTERN ************************/

externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters "{" methodPrototypes "}"
    | optAnnotations EXTERN functionPrototype ";"
    ;

methodPrototypes
    : /* empty */
    | methodPrototypes methodPrototype
    ;

functionPrototype
    : typeOrVoid name optTypeParameters "(" parameterList ")"
    ;

methodPrototype
    : optAnnotations functionPrototype ";"
    | optAnnotations TYPE_IDENTIFIER "(" parameterList ")" ";"
    ;

/************************** TYPES *************************/

typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    | p4listType
    | tupleType
    ;

namedType
    : typeName
    | specializedType
    ;

prefixedType
    : TYPE_IDENTIFIER
    | dotPrefix TYPE_IDENTIFIER
    ;

typeName
    : prefixedType
    ;
```

```
p4listType
    : LIST "<" typeArg ">"
    ;

tupleType
    : TUPLE "<" typeArgumentList ">"
    ;

headerStackType
    : typeName "[" expression "]"
    | specializedType "[" expression "]"
    ;

specializedType
    : typeName "<" typeArgumentList ">"
    ;

baseType
    : BOOL
    | MATCH_KIND
    | ERROR
    | BIT
    | STRING
    | INT
    | BIT "<" INTEGER ">"
    | INT "<" INTEGER ">"
    | VARBIT "<" INTEGER ">"
    | BIT "<" "(" expression ")" ">"
    | INT "<" "(" expression ")" ">"
    | VARBIT "<" "(" expression ")" ">"
    ;

typeOrVoid
    : typeRef
    | VOID
    | IDENTIFIER     // may be a type variable
    ;

optTypeParameters
    : /* empty */
    | typeParameters
    ;

typeParameters
    : "<" typeParameterList ">"
```

```
    ;

typeParameterList
    : name
    | typeParameterList "," name
    ;

typeArg
    : typeRef
    | nonTypeName
    | VOID
    | "_"
    ;

typeArgumentList
    : /* empty */
    | typeArg
    | typeArgumentList "," typeArg
    ;

realTypeArg
    : typeRef
    | VOID
    | "_"
    ;

realTypeArgumentList
    : realTypeArg
    | realTypeArgumentList "," typeArg
    ;

typeDeclaration
    : derivedTypeDeclaration
    | typedefDeclaration ";"
    | parserTypeDeclaration ";"
    | controlTypeDeclaration ";"
    | packageTypeDeclaration ";"
    ;

derivedTypeDeclaration
    : headerTypeDeclaration
    | headerUnionDeclaration
    | structTypeDeclaration
    | enumDeclaration
    ;
```

```
headerTypeDeclaration
    : optAnnotations HEADER name optTypeParameters "{" structFieldList "}"
    ;

structTypeDeclaration
    : optAnnotations STRUCT name optTypeParameters "{" structFieldList "}"
    ;

headerUnionDeclaration
    : optAnnotations HEADER_UNION name optTypeParameters "{" structFieldList "}"
    ;

structFieldList
    : /* empty */
    | structFieldList structField
    ;

structField
    : optAnnotations typeRef name ";"
    ;

enumDeclaration
    : optAnnotations ENUM name "{" identifierList optTrailingComma "}"
    | optAnnotations ENUM typeRef name "{"
      specifiedIdentifierList optTrailingComma "}"
    ;

specifiedIdentifierList
    : specifiedIdentifier
    | specifiedIdentifierList "," specifiedIdentifier
    ;

specifiedIdentifier
    : name "=" initializer
    ;

errorDeclaration
    : ERROR "{" identifierList "}"
    ;

matchKindDeclaration
    : MATCH_KIND "{" identifierList optTrailingComma "}"
    ;

identifierList
    : name
```

```
    | identifierList "," name
    ;

typedefDeclaration
    : optAnnotations TYPEDEF typeRef name
    | optAnnotations TYPEDEF derivedTypeDeclaration name
    | optAnnotations TYPE typeRef name
    ;

/************************** STATEMENTS ************************/

assignmentOrMethodCallStatement
    : lvalue "(" argumentList ")" ";"
    | lvalue "<" typeArgumentList ">" "(" argumentList ")" ";"
    | lvalue "="  expression ";"
    ;

emptyStatement
    : ";"
    ;

exitStatement
    : EXIT ";"
    ;

returnStatement
    : RETURN ";"
    | RETURN expression ";"
    ;

conditionalStatement
    : IF "(" expression ")" statement
    | IF "(" expression ")" statement ELSE statement
    ;

// To support direct invocation of a control or parser without instantiation
directApplication
    : typeName "." APPLY "(" argumentList ")" ";"
    | specializedType "." APPLY "(" argumentList ")" ";"
    ;

statement
    : assignmentOrMethodCallStatement
    | directApplication
    | conditionalStatement
    | emptyStatement
```

```
    | blockStatement
    | returnStatement
    | exitStatement
    | switchStatement
    ;

blockStatement
    : optAnnotations "{" statOrDeclList "}"
    ;

statOrDeclList
    : /* empty */
    | statOrDeclList statementOrDeclaration
    ;

switchStatement
    : SWITCH "(" expression ")" "{" switchCases "}"
    ;

switchCases
    : /* empty */
    | switchCases switchCase
    ;

switchCase
    : switchLabel ":" blockStatement
    | switchLabel ":"  // fall-through
    ;

switchLabel
    : DEFAULT
    | nonBraceExpression
    ;

statementOrDeclaration
    : variableDeclaration
    | constantDeclaration
    | statement
    ;

/************************ TABLE ********************************/

tableDeclaration
    : optAnnotations TABLE name "{" tablePropertyList "}"
    ;
```

```
tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY "=" "{" keyElementList "}"
    | ACTIONS "=" "{" actionList "}"
    | optAnnotations optCONST ENTRIES "=" "{" entriesList "}"
    | optAnnotations optCONST nonTableKwName "=" initializer ";"
    ;

keyElementList
    : /* empty */
    | keyElementList keyElement
    ;

keyElement
    : expression ":" name optAnnotations ";"
    ;

actionList
    : /* empty */
    | actionList optAnnotations actionRef ";"
    ;

actionRef
    : prefixedNonTypeName
    | prefixedNonTypeName "(" argumentList ")"
    ;

entry
    : optCONST entryPriority keysetExpression ':' actionRef optAnnotations ';'
    | optCONST keysetExpression ':' actionRef optAnnotations ';'
    ;

entryPriority
 : PRIORITY '=' INTEGER ":"
 | PRIORITY '=' '(' expression ')' ":"
 ;

entriesList
    : /* empty */
    | entriesList entry
    ;
```

```
/************************ ACTION ******************************/

actionDeclaration
    : optAnnotations ACTION name "(" parameterList ")" blockStatement
    ;

/************************ VARIABLES ***************************/

variableDeclaration
    : annotations typeRef name optInitializer ";"
    | typeRef name optInitializer ";"
    ;

constantDeclaration
    : optAnnotations CONST typeRef name "=" initializer ";"
    ;

optInitializer
    : /* empty */
    | "=" initializer
    ;

initializer
    : expression
    ;

/*************** Expressions ***************/

functionDeclaration
    : annotations functionPrototype blockStatement
    | functionPrototype blockStatement
    ;

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList "," argument
    ;

argument
    : expression  /* positional argument */
    | name "=" expression  /* named argument */
```

```
    | "_"
    | name "=" "_"
    ;

expressionList
    : /* empty */
    | expression
    | expressionList "," expression
    ;

structuredAnnotationBody
    : expressionList optTrailingComma
    | kvList optTrailingComma
    ;

member
    : name
    ;

prefixedNonTypeName
    : nonTypeName
    | dotPrefix nonTypeName
    ;

lvalue
    : prefixedNonTypeName
    | THIS
    | lvalue "." member
    | lvalue "[" expression "]"
    | lvalue "[" expression ":" expression "]"
    | "(" lvalue ")"
    ;

%left ","
%nonassoc "?"
%nonassoc ":"
%left "||"
%left "&&"
%left "==" "!="
%left "<" ">" "<=" ">="
%left "|"
%left "^"
%left "&"
%left "<<" ">>"
%left "++" "+" "-" "|+|" "|-|"
%left "*" "/" "%"
```

```
%right PREFIX
%nonassoc "]" "(" "["
%left "."

// Additional precedences need to be specified

expression
    : INTEGER
    | DOTS
    | STRING_LITERAL
    | TRUE
    | FALSE
    | THIS
    | prefixedNonTypeName
    | expression "[" expression "]"
    | expression "[" expression ":" expression "]"
    | "{" expressionList optTrailingComma "}"
    | "{#}"
    | "{" kvList optTrailingComma "}"
    | "{" kvList "," DOTS optTrailingComma "}"
    | "(" expression ")"
    | "!" expression %prec PREFIX
    | "~" expression %prec PREFIX
    | "-" expression %prec PREFIX
    | "+" expression %prec PREFIX
    | typeName "." member
    | ERROR "." member
    | expression "." member
    | expression "*" expression
    | expression "/" expression
    | expression "%" expression
    | expression "+" expression
    | expression "-" expression
    | expression "|+|" expression
    | expression "|-|" expression
    | expression "<<" expression
    | expression ">>" expression
    | expression "<=" expression
    | expression ">=" expression
    | expression "<" expression
    | expression ">" expression
    | expression "!=" expression
    | expression "==" expression
    | expression "&" expression
    | expression "^" expression
    | expression "|" expression
```

```
    | expression "++" expression
    | expression "&&" expression
    | expression "||" expression
    | expression "?" expression ":" expression
    | expression "<" realTypeArgumentList ">" "(" argumentList ")"
    | expression "(" argumentList ")"
    | namedType "(" argumentList ")"
    | "(" typeRef ")" expression
    ;

nonBraceExpression
    : INTEGER
    | STRING_LITERAL
    | TRUE
    | FALSE
    | THIS
    | prefixedNonTypeName
    | nonBraceExpression "[" expression "]"
    | nonBraceExpression "[" expression ":" expression "]"
    | "(" expression ")"
    | "!" expression %prec PREFIX
    | "~" expression %prec PREFIX
    | "-" expression %prec PREFIX
    | "+" expression %prec PREFIX
    | typeName "." member
    | ERROR "." member
    | nonBraceExpression "." member
    | nonBraceExpression "*" expression
    | nonBraceExpression "/" expression
    | nonBraceExpression "%" expression
    | nonBraceExpression "+" expression
    | nonBraceExpression "-" expression
    | nonBraceExpression "|+|" expression
    | nonBraceExpression "|-|" expression
    | nonBraceExpression "<<" expression
    | nonBraceExpression ">>" expression
    | nonBraceExpression "<=" expression
    | nonBraceExpression ">=" expression
    | nonBraceExpression "<" expression
    | nonBraceExpression ">" expression
    | nonBraceExpression "!=" expression
    | nonBraceExpression "==" expression
    | nonBraceExpression "&" expression
    | nonBraceExpression "^" expression
    | nonBraceExpression "|" expression
    | nonBraceExpression "++" expression
```

```
    | nonBraceExpression "&&" expression
    | nonBraceExpression "||" expression
    | nonBraceExpression "?" expression ":" expression
    | nonBraceExpression "<" realTypeArgumentList ">" "(" argumentList ")"
    | nonBraceExpression "(" argumentList ")"
    | namedType "(" argumentList ")"
    | "(" typeRef ")" expression
    ;
```