P4 v1.2 Language Specification  
(draft)

The P4.org language consortium

March 15, 2016

# Abstract

P4 is a language for programming the data plane of network devices. This document provides the formal language definition.

This document provides a precise definition of the P4 language. The main target audience for this document is composed of developers writing P4 compilers/simulators/IDEs/debuggers. The document may also be useful for P4 programmers who are interested in understanding precisely the language syntax and semantics.

Table of Contents

[1 Abstract 1](#_Toc445829976)

[Table of Contents 1](#_Toc445829977)

[2 Overview 4](#_Toc445829978)

[2.1 P4 language evolution: versions v1.0/v1.1 and v1.2 4](#_Toc445829979)

[3 Hardware-software interfaces and the Target Architecture Model 4](#_Toc445829980)

[3.1 The target architecture model 4](#_Toc445829981)

[3.2 P4 program hardware-software interfaces 4](#_Toc445829982)

[3.3 External fixed-functionality units 4](#_Toc445829983)

[4 An example: programming a simple switch 4](#_Toc445829984)

[4.1 Simple Switch architecture declaration 4](#_Toc445829985)

[4.2 Simple switch data plane architecture description 4](#_Toc445829986)

[4.2.1 The arbiter block 4](#_Toc445829987)

[4.2.2 The demux block 4](#_Toc445829988)

[4.3 A complete program for the Simple Switch 4](#_Toc445829989)

[5 P4 Language definition 4](#_Toc445829990)

[5.1 Syntax and semantics 4](#_Toc445829991)

[5.1.1 Grammar 4](#_Toc445829992)

[5.1.2 Semantics and the P4 abstract machines 4](#_Toc445829993)

[5.2 Preprocessing 4](#_Toc445829994)

[5.2.1 P4 core library 4](#_Toc445829995)

[5.3 Lexical constructs 4](#_Toc445829996)

[5.3.1 Comments 4](#_Toc445829997)

[5.3.2 Literal constants 4](#_Toc445829998)

[5.4 Naming conventions 4](#_Toc445829999)

[5.5 P4 Program structure 4](#_Toc445830000)

[5.5.1 Scopes 4](#_Toc445830001)

[5.5.2 Stateful elements 4](#_Toc445830002)

[5.6 Calling convention: call by copy in/copy out 4](#_Toc445830003)

[5.7 Paths 4](#_Toc445830004)

[5.7.1 Name resolution rules 4](#_Toc445830005)

[5.7.2 Visibility 4](#_Toc445830006)

[6 P4 data types 4](#_Toc445830007)

[6.1 Base types 4](#_Toc445830008)

[6.1.1 The void type 4](#_Toc445830009)

[6.1.2 The error type 4](#_Toc445830010)

[6.1.3 The match\_kind type 4](#_Toc445830011)

[6.1.4 Boolean 4](#_Toc445830012)

[6.1.5 Integers (signed and unsigned) 4](#_Toc445830013)

[6.2 Derived types 4](#_Toc445830014)

[6.2.1 Enumeration types 4](#_Toc445830015)

[6.2.2 Header types 4](#_Toc445830016)

[6.2.3 Header stacks 4](#_Toc445830017)

[6.2.4 Struct types 4](#_Toc445830018)

[6.2.5 Tuple types 4](#_Toc445830019)

[6.2.6 Set types 4](#_Toc445830020)

[6.2.7 Function types 4](#_Toc445830021)

[6.2.8 extern types 4](#_Toc445830022)

[6.2.9 typedef 4](#_Toc445830023)

[6.2.10 Type specialization 4](#_Toc445830024)

[6.3 Parser and control blocks types 4](#_Toc445830025)

[6.3.1 Parser type declarations 4](#_Toc445830026)

[6.3.2 Control type declarations 4](#_Toc445830027)

[7 Expressions 4](#_Toc445830028)

[7.1 Expression on error values 4](#_Toc445830029)

[7.2 Expressions on enum values 4](#_Toc445830030)

[7.3 Expressions on Boolean values 4](#_Toc445830031)

[7.3.1 Multiplexors 4](#_Toc445830032)

[7.4 Bit-string (unsigned integer) operations 4](#_Toc445830033)

[7.5 Operations on fixed-width signed integers 4](#_Toc445830034)

[7.5.1 A note about shifts 4](#_Toc445830035)

[7.6 Operations on arbitrary-precision integers 4](#_Toc445830036)

[7.7 Variable bit-string operations 4](#_Toc445830037)

[7.8 Casts 4](#_Toc445830038)

[7.8.1 Explicit casts 4](#_Toc445830039)

[7.8.2 Implicit casts 4](#_Toc445830040)

[7.8.3 Illegal arithmetic expressions 4](#_Toc445830041)

[7.9 Tuple expressions 4](#_Toc445830042)

[7.10 List expressions 4](#_Toc445830043)

[7.11 Set expressions 4](#_Toc445830044)

[7.11.1 Singleton sets 4](#_Toc445830045)

[7.11.2 The universal set 4](#_Toc445830046)

[7.11.3 Cubes 4](#_Toc445830047)

[7.11.4 Ranges 4](#_Toc445830048)

[7.11.5 Tuples 4](#_Toc445830049)

[7.12 Operations on struct types 4](#_Toc445830050)

[7.13 Operations on headers 4](#_Toc445830051)

[7.14 Expressions on header stacks 4](#_Toc445830052)

[7.15 Function calls, method invocations 4](#_Toc445830053)

[7.16 Constructor invocations 4](#_Toc445830054)

[8 Constants and variable declarations 4](#_Toc445830055)

[8.1 Constants 4](#_Toc445830056)

[8.2 Variables 4](#_Toc445830057)

[8.3 Instantiations 4](#_Toc445830058)

[9 Statements 4](#_Toc445830059)

[9.1 Assignment 4](#_Toc445830060)

[9.1.1 L-values 4](#_Toc445830061)

[9.2 The empty statement 4](#_Toc445830062)

[9.3 The block statement 4](#_Toc445830063)

[9.4 The return statement 4](#_Toc445830064)

[9.5 The conditional statement 4](#_Toc445830065)

[9.6 The switch statement 4](#_Toc445830066)

[10 Packet parsing in P4 4](#_Toc445830067)

[10.1 Parser states 4](#_Toc445830068)

[10.2 Parser declarations 4](#_Toc445830069)

[10.3 The Parser abstract machine 4](#_Toc445830070)

[10.4 Parser states 4](#_Toc445830071)

[10.5 Transition statements 4](#_Toc445830072)

[10.6 Select expressions 4](#_Toc445830073)

[10.7 Assert 4](#_Toc445830074)

[10.8 Data extraction from packets 4](#_Toc445830075)

[10.8.1 packet\_in.extract – single argument 4](#_Toc445830076)

[10.8.2 Packet.extract – two arguments 4](#_Toc445830077)

[10.8.3 packet\_in.lookahead 4](#_Toc445830078)

[10.8.4 Skipping bits 4](#_Toc445830079)

[10.9 Parsing header stacks 4](#_Toc445830080)

[10.10 Invoking sub-parsers 4](#_Toc445830081)

[10.11 Unrolling parser loops 4](#_Toc445830082)

[11 Control blocks 4](#_Toc445830083)

[11.1 Actions 4](#_Toc445830084)

[11.1.1 Invoking actions 4](#_Toc445830085)

[11.2 Tables 4](#_Toc445830086)

[11.2.1 Table properties 4](#_Toc445830087)

[11.2.2 Invoking a table (match-action unit) 4](#_Toc445830088)

[11.2.3 Match-action unit execution semantics 4](#_Toc445830089)

[11.3 The Match-Action Pipeline Abstract Machine 4](#_Toc445830090)

[12 Parameterization 4](#_Toc445830091)

[13 Packet construction (deparsing) 4](#_Toc445830092)

[13.1 Data insertion into packets 4](#_Toc445830093)

[14 Architecture description 4](#_Toc445830094)

[14.1 Example architecture description 4](#_Toc445830095)

[14.2 Target program instantiation 4](#_Toc445830096)

[14.3 A Packet Filter Model 4](#_Toc445830097)

[15 The P4 abstract machine: evaluating a P4 program 4](#_Toc445830098)

[15.1 Compile-time evaluation 4](#_Toc445830099)

[15.2 Dynamic evaluation 4](#_Toc445830100)

[15.2.1 Concurrency model 4](#_Toc445830101)

[16 Annotations 4](#_Toc445830102)

[16.1.1 Predefined annotations 4](#_Toc445830103)

[16.1.2 Target-specific annotations 4](#_Toc445830104)

[17 Appendix: P4 reserved keywords 4](#_Toc445830105)

[18 Appendix: P4 core library 4](#_Toc445830106)

[19 Appendix: checksums 4](#_Toc445830107)

[20 Appendix: P4 v1.2 grammar 4](#_Toc445830108)

# Overview

P4 is a language for expressing how packets are processed by the data plane of a programmable network forwarding element such as a programmable hardware or software switch, network interface card, router or network function appliance. While initially P4 was designed for programming network switches, its scope has been broadened to cover a large variety of network forwarding elements. In the rest of this document we use the generic term ***target*** for all such network forwarding elements.

Many target devices contain both a control plane and a data plane. P4 is designed to specify only the target data plane functionality. P4 programs may also partially define the interface by which the control plane and the data-plane communicate, but P4 **cannot** be used to describe the target’s control-plane functionality.

*In the rest of this document, when we talk about P4 as “programming a target”, we mean “programming the target data plane”.*

As a concrete example in the context or programming network switches, 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. 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.



Figure 1: Traditional switches vs. programmable switches.

A P4-programmable switch changes this equation in two ways:

* The switch data plane is no longer fixed; P4 programs describe the data plane functionality. The data plane is configured at switch initialization time based on the P4 functionality (shown by the long red arrow). The data plane has no built-in knowledge of existing protocols.
* The control plane continues to interact with the data plane using the same channels; however, the set of tables and other objects driving the behavior of the data plane is no longer fixed, since the P4 programmer specifies it long after the switch hardware has been built. The P4 compiler generates separate control-plane API for each P4 program.

P4 itself is protocol independent but allows programmers to express a rich set of data plane behaviors and protocols. The core P4 abstractions are:

* **Header definitions** describe the format (the set of fields and their sizes) of each header within a packet.
* **Parsers** describe the permitted header sequences within received packets and the headers and fields to extract from packets.
* **Tables** associate keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, or other user-defined table types, including complex multivariable decisions.
* **Actions** are code fragments that describe how packet header fields and metadata are manipulated. Actions can also include data, which can be supplied by the control-plane.
* **Match-action units** stitch together tables and actions, and 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
  + Finally, execute the selected action
* **Control flow** expresses an imperative program describing the data-dependent packet processing within a target pipeline, including the data-dependent sequence of match-action unit invocations.
* **Extern objects** are library constructs that can be manipulated by P4 program through well-defined APIs, but whose internal behavior is hardwired (e.g., checksum units).
* **User-defined metadata:** user-defined data structures associated with each packet.
* **Target-defined metadata**: metadata provided by the target architecture associated with each packet (e.g., the input port where a packet has been received).



Figure 2: Programming a target with P4

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

Target manufacturers provide the target hardware or implementation framework, a target architectural model and a P4 compiler for that target. P4 programmers write programs in P4 for a specific target architecture model. The ***target architecture model*** 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:

* code for execution in the target data plane (the ***runtime program***) and
* an API for managing the behavior of the data plane objects from the control plane

P4 is a domain-specific language. It has been designed to be portable to a large variety of target platforms such as programmable network cards, FPGA switches, software switches and programmable ASICs. As such the language is restricted to constructs that should be efficiently implementable on all these platforms. The P4 language may evolve to incorporate additional features.

P4 is not a Turing-complete language. In fact, (assuming a unit cost for a table lookup operation), all P4 program block (i.e., **parser** or **control**) should provably execute a constant O(1) number of operations for each byte of an input packet received (i.e. each header byte analyzed). In other words, the computational complexity of a P4 program depends linearly only on the header sizes, 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.

In this document the behavior of P4 programs is defined as a set of abstract machines with infinite resources. However, each target has a limited set of resources and capabilities. Rather than being the least-common denominator supported by all platforms, P4 contains powerful constructs that may be unavailable on some targets. *We expect that not all P4 programs can be compiled for all the targets.* Particular targets may not fully support some P4 language constructs (for example, some targets may not support the features necessary for IPv4 options processing or arbitrary-length stacked protocol headers). Ideally the restrictions on the P4 language imposed by a specific target should be clearly documented by the target manufacturers; at the very least, restrictions have to be conveyed to P4 programmers using clear compiler error messages when attempting to compile programs that use unsupported features. On many targets P4 programs should have an escape route for processing complex packets: they can delegate the processing of such packets to the target control plane (often referred to as the “slow path”).

***P4 conformance*** of a target is defined as follows: if a specific target T supports only a subset of the P4 programming language (let’s call it P4T), the programs written in P4T executed on the target should provide the exact same behavior as P4 programs executed on the P4 abstract machine in this document.

## P4 language evolution: versions v1.0/v1.1 and v1.2

The 1.2 version of the P4 language contains some significant backwards-incompatible changes to the language syntax and semantics. The language evolution is illustrated in Figure 3. In particular, a significant number of language constructs have been eliminated from the language and will be migrated into libraries. Examples include: counters, checksum units, meters, etc.

The original complex language (74 keywords) has been thus transformed into a relatively small core language (40 keywords) accompanied by a core library of fundamental constructs which are necessary for writing almost all P4 programs. This core language is the subject of the present specification.

The v1.1 version of P4 introduced a language construct called **extern** which can be used to describe library elements. Many constructs that are defined in the P4 v1.1 language specification will thus be transformed into such library elements (including equivalents to P4 v1.1 constructs that have been eliminated from the language, such as counters, meters). Some of these **extern** objects are expected to be standardized, and they will be in the scope of a separate document, describing a standard library of P4 elements. In this document we provide several examples of **extern** constructs.

The P4 v1.2 language version also introduces and repurposes some P4 v1.1 language constructs for describing the programmable parts of a target architecture. These language constructs are: **parser**, **state**, **control**, and **package**.



Figure 3: Evolution of the P4 language between versions 1.1 and 1.2

# Data Plane interfaces and the Target Architecture Model

## The target architecture model

***Target architecture model,*** identifies the hardware’s P4-programmable blocks (e.g., parser, ingress pipeline, egress pipeline, deparser, etc.) and their data plane interfaces.

The target architecture model can be thought of as a contract between the target and P4 code, executing on it. Each target manufacturer must provide a target architecture model for their target, and an accompanying P4 compiler. (We expect that P4 compilers for all architectures can share a common front-end). This target architecture model does not have to expose the entire programmable surface of the data plane – a manufacturer may even choose to provide multiple models for one hardware device, each with different capabilities (e.g., with or without multicast support).

Figure 4 illustrates the data plane interfaces of P4 programs. It shows a target that has two programmable blocks (#1 and #2). Each block is programmed through a dedicated P4 program fragment. 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 by *intrinsic metadata*.

Moreover, P4 programs can store and manipulate data pertaining to each packet, represented by *user-defined metadata*.



Figure 4: P4 program interfaces

The semantics of each P4 program is completely described as a set of transformations mapping structures to structures. However, only *the target architecture model attaches a meaning to the bits in a control register.* For example, in order to cause a network packet to be forwarded on a specific output port, a P4 program may need to write the output port index into a dedicated control register. Similarly, in order to cause a packet to be dropped, a P4 program may need to set a “drop” bit into another dedicated control register.

P4 programs can invoke services of fixed-function blocks. Figure 5 shows a P4 program invoking the services of a target built-in checksum computation unit. The implementation of the checksum unit is not specified in P4, but its *interface* is. Interfaces (represented by P4 **extern** objects) describe the set of operations that are offered by a fixed-function block as well as their arguments, similar to methods in object-oriented programming languages.



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

In general, P4 programs are not expected to be portable across different target architectures. For example, executing a P4 program that controls packet broadcast by writing into a custom control register will not work on a target that provides no such control register. However, P4 programs written for a given target architecture should be portable across all targets that faithfully implement the corresponding model (assuming that enough resources are available to implement the program).

With these caveats, compared to the state-of-the-art method of programming microcode on top of custom hardware, P4 provides significant advantages:

* **Flexibility**: P4 makes many packet-forwarding policies expressible as programs; contrast this to the fixed-function forwarding engines in traditional switches
* **Expressiveness:** P4 programs may express sophisticated hardware-independent packet processing algorithms using solely general-purpose registers and table look-ups. Such programs will be portable between hardware architectures that implement similar target architectures (assuming enough resources are available).
* **Resource mapping and management:** P4 programs express resource usage in symbolic terms (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage resource allocation and scheduling.
* **Software engineering:** P4 programs provide important benefits such as type checking, information hiding and software reuse.
* **Component libraries:** Manufacturer-supplied component libraries may be used to wrap hardware-specific functions into portable high-level P4 constructs.
* **Decoupling hardware and software evolution:** target manufacturers may use abstract target architectures to further decouple the evolution of low-level architectural details from high-level processing.
* **Debugging:** Manufacturers can provide their customers software models of a target architectures to aid in the development and debugging of P4 programs.
* **Standard architecture:** we expect that the P4 community will evolve a standard architecture model (or a small set of models, pertaining to specific verticals, e.g., switches and network cards). Wide adoption of such standard architectures should promote wide portability of P4 programs.

## P4 program data plane interfaces

(In the following examples P4 keywords shown in fixed-size **bold** fonts.)

To describe a functional block that can be programmed in P4 the target manufacturer provides a target prototype declaration. Target declarations are discussed in in Section14, but we give a brief introduction here to make things concrete. For example, the target manufacturer could write a declaration as follows:

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

This declaration describes a programmable block named matchActionPipe that performs match-action processing (shown by the **control** P4 reserved keyword). The interface between the matchActionPipe block and the surrounding hardware can be read from this declaration:

* **bit**<4> is a type indicating a 4-bit value,
* The **in**, **inout**, and **out** keywords indicate the direction of parameters
* Input #1: is a 4-bit value named inputPort, received in an input (**in**) control register
* Input #2: an object of type H, named parsedHeaders. It is both an input and an output, shown by **inout**.
* The type H is given by a type variable <H> in the declaration (the syntax is similar to Java). The type H indicates a type that will be defined later by the programmer. Since the type is a variable, the value parsedHeaders will be stored in general-purpose registers.
* Output #1: the same parsedHeaders is and output, also stored in general-purpose registers. The user mutates this value in-place in the pipeline implementation, and the value of the parsedHeaders at the completion of pipe is the result of the computation.
* Output #2: a four-bit value named outputPort. This value is written into a control register.

## External units with predefined functionality

P4 programs can also interact with fixed-function blocks by invoking their services. Such fixed-function blocks are described using the **extern** construct, which only describes the *interfaces* that such a block exposes to the data-plane; a complete description of **extern** is given in Section 6.2.8, but we provide an overview here.

**extern** constructs are similar to completely abstract classes from object-oriented programming (or interfaces in Java and C#), by describing a set of methods that are implemented by an object, but not the implementation of these methods. For example, the following construct could be used to describe the operations offered by an incremental checksum unit:

**extern** Checksum16 {  
 // prepare unit  
 **void** clear();  
 // add more data to be checksummed  
 **void** update<D>(**in** D data);  
 // conditionally add data to be checksummed  
 **void** update<D>(**in** **bool** condition, **in** D data);  
 // get the checksum of all data updated since the last clear  
 **bit**<16> get();  
}

# An example: programming a simple switch

To make the discussion concrete we begin by presenting an example of a simple target switch architecture, which we call the “Simple Switch” (SS). Figure 6 is a diagram of this architecture. We expect that each target manufacturer will publish target architecture models reflecting the capabilities of their own hardware – the architecture’s fixed-function blocks (shown in cyan in our example) can have arbitrarily complicated behaviors. The white blocks are programmable using P4.

SS receives the packet through one of 8 input Ethernet ports, or through a recirculation channel. SS 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

Packets may be received from one of three sources:

* One of 8 input ports
* Through recirculation
* From the control plane CPU

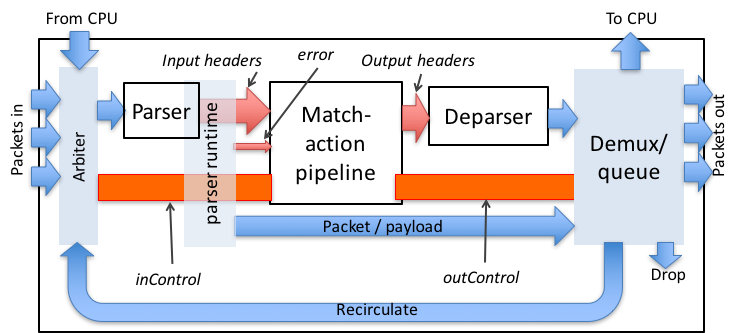


Figure 6: The Simple Switch (SS) architecture

The white blocks in the figure are programmable, and the user must provide a corresponding P4 program to drive each such block. The cyan blocks are fixed-function. The orange blocks are data plane interfaces used to convey information between the fixed-function blocks and the programmable blocks. The red arrows show the flow of user-defined data.

## Simple Switch architecture declaration

The following P4 program provides a declaration of SS, as it would be provided by the SS manufacturer. The declaration contains several type declarations, constants, and finally declarations for the programmable blocks. P4 keywords are shown in bold. The programmable blocks are given through function prototypes; the implementation of these functions has to be provided by the switch programmer.

// File “simple\_model.p4”  
// Simple switch P4 declaration

// core library needed for packet\_in definition  
#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\_INPUT\_PORT = 0xD;  
**const** PortId CPU\_INPUT\_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 parserHeaders 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 target, accompanying input packet  
 \* @param outCtrl information for target, accompanying output packet  
 \*/  
**control** Pipe<H>(**inout** H headers,  
 **in** **error** parseError, // parser error  
 **in** InControl inCtrl, // input port  
 **out** OutControl outCtrl); // output port

/\*\*  
 \* Switch deparser.  
 \* @param <H> type of headers; defined by user  
 \* @param b output packet  
 \* @param outputHeaders headers for output packet  
 **/**   
**control** Deparser<H>(**in** H outputHeaders,   
 packet\_out b);

/\*\*   
 \* Top-level package “Switch” 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** Simple<H>(Parser<H> p,   
 Pipe<H> map,   
 Deparser<H> d);

// available target-specific blocks

// Checksum unit **extern** Checksum16 {  
 **void** clear(); // prepare unit for computation  
 **void** update<T>(T data); // add data to checksum  
 **bit**<16> get(); // get the checksum for the data added since last clear  
}

Let us describe some of these elements:

* **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 some 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 a parser – but not yet its implementation. The parser implementation will have to be provided by the user. The parser reads its input from a packet\_in, which is a pre-defined P4 abstraction that represents an incoming network packet. 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 user.
* 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 3 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 Simple; in order to program a simple switch, 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** Simple<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 Simple **package** declaration has 3 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 external block whose services can be invoked to compute checksums.

## Simple switch data plane architecture description

In order to fully understand SS’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.

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 target – in the previous program this interface is given by the Parser, Pipe and Deparser declarations. In practice we expect that the complete target description will be provided as an executable program and/or diagrams and text; in this document we provide a verbal description in English.

### The 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 there is no queue space is available, input ports may drop arriving packets.
* After receiving a packet, the arbiter block sets the inCtrl.inputPort value with the identity of the input port where the packet originated. Physical Ethernet ports are numbered 0..7, while the Recirculation Port has a number 13 and the CPU port has the number 14

### The 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 amount of data unparsed) to the demux block.

### The demux block

The core functionality of the demux block is to receive the headers for the outgoing packet from the deparser, 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 forever.
* Sending the packet to an output Ethernet port numbered between 0 and 7 causes it to be emitted on the corresponding physical switch interface. The packet may be placed in a queue if the output interface is busy emitting a previous 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 CPU port causes the packet to be materialized in 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 done in a single pass.
* If the outputPort has an illegal value (e.g., 9), the packet is sent to the DROP\_PORT.
* 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 by its own choice**, 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 between the arbiter and demux blocks represents an additional information flow between arbiter and demux: the packet being processed as well as the offset within the packet where parsing ended (i.e., the start of the packet payload).

## A complete program for the Simple Switch

Here we provide a complete P4 program performing L2/L3 forwarding for IPv4 packets for the Simple Switch. This program does not take advantage of some features of the switch: e.g. recirculation. The details of many constructs will be explained throughout the document. The goal of this section is to provide an introduction to the language.



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

Parsing attempts to recognize an Ethernet header and an IPv4 header; if these headers are missing parsing terminates with an error. Otherwise it extracts the information from these headers into a structure with type Parsed\_packet.

The match-action pipeline is shown in Figure 7; it comprises 4 match-action units (represented by the P4 **table** keyword):

* If any parser error has occurred, the packet is dropped (sending it to DROP\_PORT)
* The first **table** uses the IPv4 destination address to discover the outputPort and the IPv4 address of the next hop. If this look-up fails, the packet is dropped
* The second **table** decrements the IPv4 ttl. If the ttl becomes 0, the packet is sent through the CPU port towards the control plane.
* The third **table** uses the IPv4 address of the next hop (computed by the first **table**) to figure out the Ethernet address of the next hop.
* Finally, the last **table** uses the outputPort to discover the source Ethernet address of the current switch.

The deparser puts together a packet by reassembling the Ethernet and IPv4 headers as computed in the pipeline.

This example uses the C preprocessor to execute the #include directive.

#include “core.p4”

// include the switch declaration from the previous section

#include “simple\_model.p4”

// This program processes packets composed of an Ethernet and

// an IPv4 header, performing forwarding based on 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;  
}  
   
// Parser section  
   
// Declare user-defined errors that may be signaled during parsing  
**error** {   
 IPv4OptionsNotSupported,  
 IPv4IncorrectVersion,

IPv4ChecksumError  
}

// List of all recognized headers  
**struct** Parsed\_packet {  
 Ethernet\_h ethernet;  
 Ipv4\_h ip;  
}  
   
**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);  
 **assert**(p.ip.version == 4w4, IPv4IncorrectVersion);  
 **assert**(p.ip.ihl == 4w5, IPv4OptionsNotSupported);  
 ck.clear();  
 ck.update(p.ip);

// Verify that packet checksum is zero  
 assert(ck.get() == 16w0, IPv4ChecksumError);  
 **transition** accept;  
 }  
}

// match-action pipeline section

**control Top**Pipe(**inout** Parsed\_packet headers,  
 **in** **error** parseError, // parser error  
 **in** InControl inCtrl, // input port  
 **out** OutControl outCtrl) {  
 /\*\*

\* Indicates that a packet is dropped by setting the   
 \* output port to the DROP\_PORT  
 \*/  
 **action** Drop\_action()   
 { outCtrl.port = DROP\_PORT; }

**/**\*\*   
 \* Set the next hop and the output port.  
 \* Decrements ipv4 ttl field.  
 \* @param ivp4\_dest ipv4 address of next hop  
 \* @param port output port   
 \*/  
 **action** Set\_nhop(**out** IPv4Address nextHop,  
 Ipv4Addres 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(**out** Ipv4Address nextHop) **{** key = { headers.ip.dstAddr : lpm; }  
 actions = {  
 Drop\_action; Set\_nhop(nextHop); }

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(**in** Ipv4Address nextHop) {  
 key = { nextHop : exact; }   
 actions = {  
 Drop\_action;  
 Set\_dmac;  
 }  
 size = 1024;  
 default\_action = Drop\_action;  
 }

/\*\*

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

/\*\*

\* Set the source mac address based on the output port.  
 \*/  
 **table** smac() {  
 key = { outCtrl.outputPort : exact; }  
 actions = {  
 Drop\_action  
 Rewrite\_smac;  
 }

size = 16;  
 default\_action = Drop\_action;  
 }

**apply** {  
 // match-action pipeline body  
 **if** (parseError != NoError) {  
 // invoke drop directly  
 Drop\_action();  
 **return**;  
 }  
   
 Ipv4Address nextHop; // temporary variable  
 ipv4\_match.apply(nextHop); // write result in nextHop  
 **if** (outCtrl.outputPort == DROP\_PORT)  
 **return**;  
 check\_ttl.apply();  
 **if** (outCtrl.outputPort == CPU\_PORT)  
 **return**;  
 dmac.apply(nextHop);  
 **if** (outCtrl.outputPort == DROP\_PORT)  
 **return**;  
 smac.apply();

} // end of apply  
} // end of control

// deparser section  
**control** TopDeparser(**in** Parsed\_packet p, packet\_out b) {

Checksum16() ck;

**apply** {  
 b.emit(p.ethernet);

if (p.ip.valid) {

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 Simple package.

// use TopParser for the p Parser, etc.  
Switch(TopParser(),  
 TopPipe(),  
 TopDeparser()) main;

# 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 finite-state machines (Section 10).
* A sub-language for expressing **match-action** computations, based on traditional imperative control-flow (Section 11).
* A sub-language for describing target architectures (Section 14).

## Syntax and semantics

### Grammar

The complete grammar of P4 v1.2 is given in Appendix 20, using the YACC/bison grammar description language. In this text we use the same grammar; we use the following conventions when we provide excerpts from the grammar:

* grammar rules are written using fixed-size font
* UPPERCASE symbols denote grammar terminals

Grammar fragments will be shown on a cyan background, as in the following example:

p4program

: /\* empty \*/

| p4program declaration

;

P4 code is written using a fixed-size font on a white background.

Pseudo-code examples (mostly used for describing the semantics of various P4 constructs) are shown with fixed-size fonts on a light-green background, as in the following example:

ParserModel.assert(bool condition, error err) {  
 if (eval(condition) == false) {  
 ParserModel.parserError = eval(err);  
 ParserModel.currentState = reject;  
 }  
}

### Semantics and the P4 abstract machines

The P4 semantics is described 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.

## Preprocessing

P4 does not support separate compilation or linking. The entire program must be presented to the compiler as a single input file.

To aid in modularization, P4 language supports the following subset of the C preprocessor functionality:

* #define for defining macros without arguments
* #undef
* #if/#else/#endif/#ifdef
* #include

Additional capabilities of the C preprocessor may be supported, but are not guaranteed (e.g., macros with arguments).

In addition to that, P4 compilers should correctly handle #line directives that may be generated by the C preprocessor.

This functionality allows P4 programs to be built from multiple source files, potentially produced by different programmers at different times:

* the P4 core library, produced by the P4 language designers
* the target architecture interfaces, specified by the target manufacturer
* target libraries, describing **extern** blocks provided by the target architecture
* user-defined and other libraries of useful components (e.g. standard protocol header definitions)
* the P4 programs that controls programmable functional blocks of a target

### P4 core library

Similar to the C standard library, the P4 language definition mandates a core library, which declares useful P4 constructs. A description of the P4 core library is provided in the Appendix 18. All P4 programs must include the core library.

## Lexical constructs

P4 uses only ASCII characters.

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-like indentation. Tab is treated as a space.

P4 identifiers may contain only letters, numbers and the underscore sign \_, and must start with a letter or with underscore. The special identifier consisting of a single underscore \_ is reserved to indicate a “don’t care” value in several contexts; its type may vary depending on the context.

The lexer recognizes the following kinds of terminals:

* IDENTIFIER – start with a letter, and contain letters, digits and underscores
* TYPE – identifier that denotes a type name
* NAMESPACE – identifier that denotes a namespace
* INTEGER – integer literals
* Keywords, e.g., RETURN. 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

### Comments

Comments are Java style:

* Single-line comments, spanning to the end of line, introduced by //
* Multi-line comments, enclosed between /\* and \*/. Nested multi-line comments are **not** supported.
* JavaDoc comments, starting with /\*\* and ending with \*/. JavaDoc comments are strongly encouraged for the language P4 elements that are used to synthesize the interface with the control-plane: **table**s and **action**s.

Comments are tokens separators: no comments are allowed within a token, e.g.

bi/\*\*/t

is *not* parsed as a **bit** token.

### Literal constants

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

Numeric constants are integers of an arbitrary precision, positive or negative. By default, literals are assumed in base 10. To use a different base for the literal, one of the following prefixes must be employed:

* 0x or 0X indicates base 16
* 0o or 0O indicates base 8
* 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

The underscore character is considered a digit within number literals (where it cannot appear as the first digit), but it is ignored when computing the value of the parsed number. This allows long constants numbers to be more easily read by grouping digits together. No comments or whitespaces are allowed within a literal. Here are some examples of numeric literals:

32w0xFF // a 32-bit unsigned number with value 255  
32s0xFF // a 32-bit signed number with value 255  
-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

## Naming conventions

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

In this document we adopt the Java-like naming guidelines:

* the P4 built-in types all start with lowercase characters and are shown in bold, e.g., **int**<20>
* In all our examples we write user-defined types with an uppercase character, e.g., Ipv4Address
* Type variables (e.g., used in templates) are always uppercase, e.g., **parser** P<H, IH>(...)
* All variables are named with lowercase names, e.g., ipv4header
* Constants are all uppercase, e.g., CPU\_PORT
* Errors and enumerations have camel-case names, e.g. PacketTooShort

## P4 Program structure

A P4 program is a collection of declarations:

p4program

: /\* empty \*/

| p4program declaration

;

declaration

: constantDeclaration

| externDeclaration

| actionDeclaration

| parserDeclaration

| typeDeclaration

| controlDeclaration

| instantiation

| errorDeclaration

| matchKindDeclaration

;

### Scopes

Various constructs act as namespaces, that create local scopes for names:

* Derived type declarations (**struct**, **header**, **enum**) introduce local scopes for the field names
* Block statements introduce local lexically-enclosed scopes
* **parser**, **table**, **action**, and **control** blocks introduce local scopes
* A declaration with type variables introduces a new scope for that variable. For example, in the following **control** declaration, the scope of the type variable H extends to the end of the declaration:  
  **control** c<H>( ... ) { ... } // 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 change from the P4 v1.1 specification, which allows declarations in any order.)

### Stateful elements

Most P4 constructs are *stateless*: given some inputs they produce a result that solely depends on these inputs. There are only two kinds of constructs that may retain information across packets (we call them “stateful”):

* **table**s. Tables are read-only for the data plane; their contents can only be modified by the control-plane
* **extern** objects. Some of these objects may be both read and modified by the data plane. All constructs from the P4 v1.0 language version that represent state (counters, meters, registers) are represented using **extern** objects in P4 v1.2.

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

Both **parser**s, **control** blocks and **package**s may contain stateful elements instantiations; thus they are also treated as stateful elements (even if they happen to contain no actual state). All stateful elements (**extern** objects, **parser**s, **control**s, **package**s, but not **table**s) are represented in P4 by *types*. In order to use such an object, it must be first instantiated.

Considering the example in Section 4.3, TopParser, TopPipe, TopDeparser, Checksum16 and Switch are types. These are instantiated in the program to be used: there are two instances of Checksum16, one in TopParser and one in TopDeparser, both called ck. The TopParser, TopDeparser, TopPipe and Switch are instantiated in the end or the program, in the declaration of the main variable, which is an instance of the Switch type (a package).

All global variables in P4 must be stateful objects.

## Calling convention: call by copy in/copy out

P4 provides multiple constructs for writing modular programs: **extern** methods, **parser**s, **action**s, **table**s. etc. All these constructs behave similarly to procedures in traditional 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 done using a copy-in/copy-out semantics.

Each parameter is labeled with a direction:

* **in** parameters are read-only. It is an error if an **in** parameter is used on the left-hand side of an assignment or is passed as a non-**in** argument to a callee. **in** parameters are always initialized by copying the value of the corresponding argument at call execution time.
* **out** parameters are uninitialized; they are l-values (they can be assigned to – l-values are described in Section 9.1.1.). The corresponding argument in a call must be an l-value; after execution of a call, the value of an **out** parameter is copied out to the corresponding argument.
* **inout** parameters are both **in** and **out**. They must be bound to an l-value argument.
* No direction indicates that value of parameter is either a compile-time constant, or an action parameter that can only be set by the control plane.

At invocation time arguments are evaluated in order, from left to right; all **in** and **inout** arguments are evaluated prior to the callee execution starting. After the callee execution terminates, the values of all **out** and **inout** parameters are copied to the corresponding arguments.

parameterList

: /\* empty \*/

| nonEmptyParameterList

;

nonEmptyParameterList

: parameter

| nonEmptyParameterList ',' parameter

;

parameter

: optAnnotations direction typeRef name

;

direction

: IN

| OUT

| INOUT

| /\* empty \*/

;

name

: IDENTIFIER

| APPLY

| KEY

| ACTIONS

| STATE

;

This calling convention raises the problem of argument aliasing, where two arguments point to the same underlying object. The compiler should detect and flag as a warning if multiple arguments to the same object refer to the same underlying storage and any of them are out or inout.

When used as arguments, **extern** objects can only be passed as directionless parameters (see for example the packet argument in the simple switch example).

## Paths

A path prefix is a sequence of namespaces separated by dots. A leading dot indicates an absolute path.

pathPrefix

: '.' relativePathPrefix

| nonEmptyRelativePathPrefix

;

relativePathPrefix

: /\* empty \*/

| relativePathPrefix NAMESPACE '.'

;

nonEmptyRelativePathPrefix

: relativePathPrefix NAMESPACE '.'

;

All paths that start with a dot are absolute paths, otherwise they are relative paths (interpreted relative to the current namespace in scope). For example, the following are legal paths:

Sw.Pkg.Ingress // relative path  
.Sw.Pkg.Ingress // absolute path

### Name resolution rules

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

Absolute paths are always resolved in the top-level namespace.

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** {}  
}

### Visibility

Identifiers defined in the top-level namespace are globally visible.

Currently all declarations in an inner namespace are private to that namespace.

# P4 data types

P4 is a strongly-typed language; all values are statically typed. Programs that do not pass type-checking are invalid. Some values can be converted to a different type by using casts. To make the user intent clear, implicit casts are allowed on in very few circumstances. Moreover, the range of casts available is intentionally restricted.

P4 supports several base types and allows the construction of derived types.

## Base types

P4 supports the following built-in base types:

* The **void** type, which has no values (can be used only in restricted circumstances)
* The **error** type, used to convey errors in a machine-independent, compiler-managed way
* Boolean values
* Bit-strings of fixed width
* Bit-strings of dynamically-computed width with a fixed maximum width
* Fixed-width signed integers represented using two’s complement

baseType

: BOOL

| ERROR

| BIT

| BIT '<' INTEGER '>'

| INT '<' INTEGER '>'

| VARBIT '<' INTEGER '>'

;

### The void type

The void type is written as **void.** It contains no values. It can only appear in few restricted places in P4 programs.

### The error type

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

errorDeclaration

: ERROR '{' identifierList '}'

;

All error constants thus declared are inserted in the top-level namespace, irrespective of the place where an error is defined. **error** is similar to a C/C# enumeration (enum) type. A program can contain multiple **error** declarations, which the compiler will merge together.

For example, the following declaration creates two constants of **error** type (these declarations are from the P4 core library):

**error** { ParseError, PacketTooShort }

The underlying representation of errors is target-dependent.

### The match\_kind type

The **match\_kind** type is very similar to the **error** type, and to a C enum type. It is used to declare a set of names that may be used in a **table**’s key property (described in Section 11.2.1.1).

matchKindDeclaration

: MATCH\_KIND '{' identifierList '}'

;

The core library contains the following **match\_kind** declaration:

**match\_kind** {  
 exact,  
 ternary,  
 lpm  
}

Target architectures may provide tables that support additional **match\_types**.

### Boolean

The Boolean type contains two values, **false** and **true**. The type is written as**bool**. Booleans are not integers.

### 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 from 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 remedy the undefined C behaviors. Unfortunately, C has many undefined behaviors, including specifying the size of an integer (int), what results produced are on overflow, and the results produced for some input combinations (e.g., shifts with negative amounts, division of negative numbers, overflows on signed numbers, etc.). In contrast, 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 also chosen identical to C (e.g., multiplication binds stronger than addition).

### Strings

P4 offers no support for string processing. The only strings that can appear in a P4 program are constant string literals. String literals can only be used in annotations (described in Section 16). For example, the following annotation could be used to indicate that a specific name should be used for a table when generating the control-plane API:

@name(“acl”) **table** t1() { … }

#### Portability

No P4 target can support all possible types and operations. For example, the following type is legal in P4: **bit**<23132312>, but it is highly unlikely to be supported by any practical targets. 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 with small constants, or shifts with small values).

Target-specific documentation should describe such 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.

#### 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 a compile-time constant value evaluating to a positive integer greater than 0.

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 target architectures may impose additional compile-time or runtime 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 7.4.

#### Signed Integers

Signed integers are represented using 2’s complement. An integer with W bits is declared as: **int**<W>. W must be a compile-time constant value evaluating to a positive integer greater than 1

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 target architectures may impose additional compile-time or runtime 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 7.5.

#### 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**.

**varbit**<W> denotes a bit-string with a width of at most W bits, where W is a compile time constant value evaluating to a positive 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.

P4 target architectures may impose additional compile-time or runtime 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 7.6.

#### Infinite-precision integers

The infinite-precision datatype describes integers with an unlimited precision. This type is written as **int**. This type is reserved for compile-time integer literals only. No P4 run-time 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 infinite-precision integers are described in Section **7.6**.

#### Integer literal types

The types of integer literals (constants) are as follows:

* A simple integer constant has type **int**.
* A positive integer prefixed with an integer width N and the character w has type **bit**<N>.
* An integer prefixed with an integer width N and the character s has type **int**<N>.

The table below shows several examples of integer literals and their types. For additional examples of literals see Section 5.3.2.

|  |  |
| --- | --- |
| Literal | Interpretation |
| 10 | Type is **int**, value is 10 |
| -10 | Type is **int**, value is -10 |
| 8w10 | Type is **bit**<8>, value is 10 |
| -8w10 | Type is **bit**<8>, value is 245, warning for negative unsigned value |
| 8s10 | Type is **int**<10>, value is 10 |
| -8s10 | Type is **int**<10>, 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 |
| 1s10 | Type is **bit**<1>, value is 0 (last bit), overflow warning |

## Derived types

Additional types can be created in P4 using type constructors. P4 provides the following type constructors:

* **enum**
* **header**
* **struct**
* header stacks
* type specialization

**header**, **enum**, and **struct** 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.

typeDeclaration

: derivedTypeDeclaration

| typedefDeclaration

| parserTypeDeclaration ';'

| controlTypeDeclaration ';'

| packageTypeDeclaration ';'

;

derivedTypeDeclaration

: headerTypeDeclaration

| structTypeDeclaration

| enumDeclaration

;

typeRef

: baseType

| typeName

| DONTCARE

| specializedType

| headerStackType

;

typeName

: TYPE

| pathPrefix TYPE

;

### Enumeration types

An enumeration type is a more restricted version of the C enum. It is defined with the following syntax:

enumDeclaration

: optAnnotations ENUM name '{' identifierList '}'  
 ;

identifierList

: name

| identifierList ',' name

;

Annotations, represented by the non-terminal optAnnotations are described in Section 16.

This declaration introduces a new identifier in the current scope for naming the created type. The underlying representation of such values is not specified, so their “size” in bits is not specified (it is target-specific). Operations on **enum** values are described in Section 7.2.

### Header types

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

headerTypeDeclaration

: optAnnotations HEADER name '{' structFieldList '}'

;

structFieldList

: /\* empty \*/

| structFieldList structField

;

structField

: optAnnotations typeRef name ';'

| optAnnotations typeRef TYPE ';' /\* fields named like types \*/

;

where each typeRef is restricted to a bit-string type (fixed or variable) or an integer type. This 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. In addition, a header also contains a pre-defined Boolean field named valid. (The valid name is reserved, and cannot be used in a user-declared field.) When this bit is **true** we say that the “header is valid”. When a header is created its valid bit is automatically set to **false**. An empty header is acceptable:

**header** Empty\_h { }

Note that an empty header still contains a valid bit.

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 a compile-time constant, and it is simply the sum of the sizes of all component fields. There is no padding or alignment of the header fields. *Individual P4 targets may impose some constraints on header types, e.g., restricting headers to sizes that are an integral 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 type can be referred to using the introduced identifier; the following is a variable declaration using the newly introduced type:

Ethernet\_h ethernetHeader;

The P4 parser language uses the extract method of a packet to “fill in” the fields of a header from a network packet, as described in Section 10.8. The successful execution of an extract operation also sets the valid 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 discussed in Section 7.9, headers that contain variable-length fields may need to be parsed in multiple steps by being broken into multiple headers.

### Header stacks

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

headerStackType

: typeName '[' expression ']'

;

where typeName is the name of a header type. For a header stack H[*W*] *W* is the maximum defined size, and it must be a compile-time constant that evaluates to a positive integer. Nested header stacks are not supported. At run-time a stack contains *W* values with type typeName, only some of which may be valid. Expressions manipulating header-stacks are discussed in Section 7.9; header-stack specific statements are discussed in Section 11.1.1.

For example, the declarations below:

**header** Mpls\_h {  
 **bit**<20> label;  
 **bit**<3> tc;  
 **bit** bos;  
 **bit**<8> ttl;  
}

Mpls\_h[10] mpls\_vec;

introduce a header stack called mpls\_vec containing 10 entries, each of type Mpls\_h.

### Struct types

P4 **struct** types are similar to C/C++ struct types. They are defined with the following syntax:

structTypeDeclaration

: optAnnotations STRUCT name '{' structFieldList '}'

;

This declaration introduces a new type with the specified name in the local scope. An empty **struct** is legal.

For example, the structure parsed\_headers below contains the headers supported by a simple parser:

**header** Tcp\_h { … }  
**header** Udp\_h { … }

**struct** Parsed\_headers {  
 Ethernet\_h ethernet;  
 Ip\_h ip;  
 Tcp\_h tcp;  
 Udp\_h udp;  
}

### Tuple types

A tuple is similar to a **struct**, in that it holds multiple values. Unlike a **struct** types, tuples types cannot be defined explicitly, they have no names and no named fields. Tuple types are synthesized by the compiler internally. We denote a the type of tuples with n component types *T1, …, Tn* by tuple<*T1,…, Tn*>. Please note that this is not P4 syntax, it is just a notation that we use in this document to simplify the explanations. Operations that manipulate tuple types are described in Section 7.9.

### Set types

set<T> is a type that describes ***sets*** of values of 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; it can be used as a label in a **select** expression, matching any value in this range. Set types cannot be named or declared by P4 programmers, they are only synthesized by the compiler internally. Expressions with set types are described in Section 7.11.

### Function types

Currently function types cannot be created explicitly in P4 programs; they are created by the P4 compiler internally to represent the type of a function, procedure, or method for type-checking. We also call the type of a function its *signature*. Libraries can contain extern function declarations.

For example, the assert function from the P4 core library has the following signature:

**extern void** assert(**in** **bool** condition, **in** **error** errCode);

We say that assert’s type is a “function type”, representing the following information:

* the result type is **void**
* the function has two inputs
* first input has direction **in**, type **bool** and name condition
* second input has direction **in**, type **error**, and name errCode

### extern types

P4 programs can invoke the service of fixed-function blocks. A typical example of such a fixed-function block is a checksum unit. The functionality of such blocks is exposed to P4 programs through **extern** declarations.

P4 supports extern object declarations and extern function declarations.

externDeclaration

: EXTERN name optTypeParameters '{' methodPrototypes '}'

| EXTERN functionPrototype

;

#### Extern functions

An extern function declaration describes a function and its signature, but not the function’s implementation.

functionPrototype

: typeOrVoid name optTypeParameters '(' parameterList ')' ';'

;

#### 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, and no return type. Extern declarations can only appear in libraries.

methodPrototypes

: /\* empty \*/

| methodPrototypes methodPrototype

;

methodPrototype

: functionPrototype

| TYPE optTypeParameters '(' parameterList ')' ';' //constructor

;

typeOrVoid

: typeRef

| VOID

| name // may be a type variable

;

optTypeParameters

: /\* empty \*/

| '<' typeParameterList '>'

;

typeParameterList

: name

| typeParameterList ',' name

;

For example, the P4 core library introduces two interfaces packet\_in and packet\_out used for manipulating network packets (see Sections 10.8 and 13.1). Here is an example showing how operations on a network packet can be invoked:

**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 of byte stream  
}

Functions and methods are the only P4 constructs which support overloading: there can exist multiple methods with the same name in the same scope. Even so, two functions (or methods of an **extern** object) can have the same name only if they have a different number of parameters.

### typedef

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

typedefDeclaration

: TYPEDEF typeRef optAnnotations name ';'

| TYPEDEF derivedTypeDeclaration optAnnotations name ';'

;

**typedef** bit<32> u32;  
**typedef** **struct** Point { **int**<32> x; **int**<32> y; } Pt;  
**typedef** Empty\_h[32] HeaderStack;

All operations that can be executed on the original type can be also executed on the newly created type. This behavior is similar to the C typedef keyword.

### 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

: pathPrefix TYPE '<' typeArgumentList '>'

| TYPE '<' typeArgumentList '>'

;

For example, the following **extern** declaration describes a generic block or 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.

## 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. Parsers and control block types may be generic (i.e., have type parameters):

### 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 depends on a type variable IH. The parser that receives as input a packet\_in value b and produces two values:

* A value with a user-defined type IH
* A value with a predefined type Counters

**struct** Counters { … }  
**parser** P<IH>(packet\_in b,   
 **out** IH packetHeaders,   
 **out** Counters counters);

### Control type declarations

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

controlTypeDeclaration

: optAnnotations CONTROL name optTypeParameters   
 '(' parameterList ')'

;

Control type declarations are very similar to parser type declarations.

# Expressions

This section describes all computations that can be performed in P4, grouped by the type of the values than can be processed.

The grammar for general expressions is given by:

expression

: INTEGER

| TRUE

| FALSE

| name

| pathPrefix name

| expression '[' expression ']'

| expression '[' expression ':' expression ']'

| '{' expressionList '}'

| '(' expression ')'

| '!' expression

| '~' expression

| '-' expression

| '+' expression

| typeName '.' member

| expression '.' member

| 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

| 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 '<' typeArgumentList '>' '(' argumentList ')'

| expression '(' argumentList ')'

| typeRef '(' argumentList ')'

| '(' typeRef ')' expression

;

expressionList

: expression

| expressionList ',' expression

;

member

: name

;

argumentList

: /\* empty \*/

| nonEmptyArgList

;

nonEmptyArgList

: argument

| nonEmptyArgList ',' argument

;

argument

: expression

;

typeArgumentList

: typeRef

| typeArgumentList ',' typeRef

;

See also the complete P4 grammar in Appendix 20.

An additional semantic check is required for the right shift to check that there is no space between the two consecutive greater-than signs > >. This rule is required to allow parsing for both the right shift operators and specialized types, such as in function<**bit**<32>>.

This grammar does not indicate the precedence of the various operators. The precedence follows exactly the C precedence rules. 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, **select** expressions (described in Section 10.6) may be used only in parsers.

## Expression on error values

The **error** type only supports comparisons for equality and difference. The result of a comparison is a Boolean value.

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

**error** errorFromParser;  
…  
if (errorFromParser != NoError) { ... }

## Expressions on enum values

Symbolic names declared by an **enum** do not belong to the global namespace, but to a newly introduced namespace:

**enum** X { v1, v2, v3 }

X.v1 // reference to v1  
v1 // error – v1 is not in the top-level namespace

**enum** values can only be compared for equality/difference using == and !=. **enum** values cannot be cast to or from any other types.

When **enum** values appear in the control-plane API the compiler back-end has to choose a suitable serialization format.

## Expressions on Boolean values

The following operations are provided on Boolean values:

* And, designated by &&
* Or designated by ||
* Negation, designated by !
* Equality tests (== and !=)

Operator precedence is similar to C. Operator evaluation is short-circuit.

There are no implicit casts from bit-strings to Booleans or vice-versa. In consequence, a C program fragment such as:

if (x) ...

(for x an integer base type) must be written in P4 as:

if (x != 0) ...

(see also the discussion on infinite-precision types and implicit casts in Section 7.8 for how the 0 in this expression is evaluated).

### Multiplexors

The ?: expression behaves as in C, e.g.:

(x == 0) ? e0 : e1;

The first argument is Boolean, and the second and third arguments must have the same type. The second and third arguments cannot be both infinite precision integers.

## Bit-string (unsigned integer) operations

This section discusses all operations that can be performed on values with **bit**<W> types.

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). There are no arithmetic exceptions; the runtime result of an arithmetic operation is defined for all combinations of input arguments.

All binary operations (except shifts) require both operands to have the same exact type and width; supplying operands with different widths produces a compile-time error. No implicit casts are inserted by the compiler to equalize the widths. There are no binary operations that combine signed and unsigned values (except shifts).

The following operations are provided on Bit-string values:

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

All the following operations produce bit-string results when applied to bit-strings. All these operations require both operands to have the same width.

* Negation, denoted by unary –. Result is computed by subtracting the value from 2W. The result is always unsigned and it has the same width as the input. The semantics is the same as the C negation of unsigned numbers.
* Unary plus, denoted by +. Behaves as a no-op.
* Addition, denoted by +. Associative. Result is computed by truncating the result of the addition to the width of the output (similar to C).
* Subtraction, denoted by –. An associative operation. Result is unsigned, and has the same type as the operands. Result is computed by adding the negation of the second operand (similar to C).
* Bitwise “and” between two bit-strings of the same width, designated by &
* Bitwise “or” between two bit-strings of the same width, designated by |
* Bitwise “complement” of a single bit-string, designated by ~
* Bitwise “xor” of two bit-strings of the same width, designated by ^

There are also the following operations:

* Concatenation of two bit-strings, resulting in a bit-string whose length is the sum of the lengths, designated by the infix operator ++. The left bit-string provides the most significant bits.
* Extraction of a set of contiguous bits, designated by [*m*:*l*], where *m* and *l* are compile-time constants, and *m* >= *l*. The result is a bit-string of width *m – l* + 1, including the bits numbered from *l* (which becomes the LSB of the result) to *m* (the MSB of the result) from the source operand. The conditions *0<= l < W* and *l <=* *m < W* are checked statically (*W* is the length of the source bit-string). Note that both endpoints of the extraction are inclusive. The bounds must be constant so that the result width must be computed at compilation time.
* Logical shift left and right with a runtime known unsigned integer value (left operand is unsigned, right operand must be either an unsigned number of type **bit**<S> or a non-negative constant integer), designated by << and >>. The result has the same type as the left operand. Shifts with an amount greater than the width of the input produce a result with all bits zero.

## Operations on fixed-width signed integers

This section discusses all operations that can be performed on **int**<W> types. An **int**<W> type is a signed integer with W bits represented using 2’s complement.

“Underflow” or “overflow” produced by arithmetic cannot be detected: operations “wraparound”, 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)5. There are no arithmetic exceptions; the runtime result of an arithmetic operation is defined for all combinations of input arguments (note that C does not specify the result of overflows on signed integers).

All binary operations (except shifts) require both operands to have the same exact type (signedness) and width; supplying operands with different widths or signedness produces a compile-time error. No implicit casts are inserted by the compiler to equalize the types. There are no binary operations that combine signed and unsigned values (except shifts).

Note that bitwise operations are well-defined, since the representation is mandated to be 2’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 +. Behaves as a no-op.
* Addition, denoted by +
* Subtraction, denoted by –
* Comparison for equality and inequality ==, != producing a Boolean result
* Numeric comparisons <, <=, >, >= with a Boolean result
* Multiplication \*. Result has the same width as the operands. P4 targets may impose additional restrictions (e.g., may only allow multiplications with powers of two).
* Arithmetic shift left and right denoted by << and >>. Left operand is signed, right operand must be either an unsigned number of type bit<S> or a non-negative constant integer. The result has the same type as the left operand. Shifts with an amount greater or equal to the width of the input are allowed.

### A note about shifts

Shifts (on signed and unsigned values) deserve a special discussion for the following reasons:

* As in C, right shift behaves differently for signed and unsigned values: right shift for signed values is an arithmetic shift.
* Shifting with a negative amount does not have a clear semantics: while in C the result is undefined, in P4 the type system makes it illegal to shift with a negative amount.
* In C, shifting with an amount larger or equal to the number of bits has an undefined result (unlike the P4 definition).
* Finally, 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;

... y << x ...  
... y << 1024 ...

Unlike C, P4 gives a precise meaning shifting with an amount larger than the size of the shifted value.

P4 targets may impose additional restrictions on shift operations:

* Targets may reject shifts by non-constant amounts.
* Targets may reject shifts with large non-constant amounts. For example, a target may forbid shifting an 8-bit value by a non-constant value wider than 3 bits.

## Operations on arbitrary-precision integers

The type **int** denotes integer values on which computations are performed with arbitrary precision. The only values that can have the type int are compile-time constants. They support the following operations:

* Negation, denoted by unary –
* Unary plus, denoted by +. Behaves as a no-op.
* Addition, denoted by +
* Subtraction, denoted by –
* Comparison for equality and inequality ==,!= producing a Boolean result
* Numeric comparisons <, <=, >, >= with a Boolean result
* Multiplication \*
* Integer division between positive values, denoted by /, rounded towards 0, as in C
* Modulo between positive values, denoted by %
* Arithmetic shift left and right denoted by << and >>. Right operand must be a positive number. The result is an **int**. a << b is . a >> b is

All the operands that participate in an operation must have type **int**; binary operations (except shift) cannot combine **int** values with fixed-width types. For such expressions the compiler will always insert an implicit cast; this cast will always convert the **int** value to the fixed-width type.

All computations on **int** values are carried without information loss. 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 the truncation causes significant bits to be lost, the compiler should emit a suitable warning.

Note: bitwise-operations (|, &, ^, ˜) are not defined for **int** values. Division and modulo are illegal for negative values (the C language does not give a clear semantics to division of signed integers when values are negative).

## Variable bit-string operations

A variable-size bit-string has a maximum size static declared width, and also a dynamic width, which must be smaller or equal than the static width. Prior to initialization a variable-size bit-string has an unknown dynamic width.

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

* Parser extraction into a variable-sized bit-string using the two-argument extract method of a packet\_in (see Section 10.8.2). This operation sets the dynamic width of the field.
* Assignment to another variable-sized bit-string. The target must have either the exact same static width. The assignment sets the dynamic width on the target to be the same as the source dynamic width.
* the emit method of a packet\_out interface to insert a dynamically-sized bit-string with known dynamic width into a packet (see Section 13.1).

## Casts

P4 supports a very limited range of casts. Most casts must be explicit. Most binary operations require both operands to have the exact same type. Some type conversions may require multiple chained casts. While more onerous for the user, this approach has several benefits:

* It makes user intent unambiguous.
* It makes the conversion cost explicit. Some casts involve sign-extensions, and thus require significant computational resources.
* It reduces the number of cases that have to be considered in the P4 specification. Some targets may not support all casts. A cast expression is written as in C: (typeRef)

### Explicit casts

Here are all legal casts:

* **bit**<1> <-> **bool**: 0 is **false**, 1 is **true**
* **int**<*W*> -> **bit**<*W*>: preserves all bits unchanged; negative values are reinterpreted as positive values
* **bit**<*W*> -> **int**<*W*>: preserves all bits unchanged; values with the MSB set are reinterpreted as negative values
* **bit**<*W*> -> **bit**<*X*>: if *W > X* this causes truncation, if *W < X* this causes extension with zero bits
* **int**<W> -> **int**<X>: if *W > X* this causes truncation, if *W < X* this causes extension with the sign bit
* **int** -> **bit**<*W*>: Represents the integer value using two’s complement on a large enough number of bits and keeps the least-significant *W* bits; overflow should lead to a warning, as will conversion of a negative number
* **int -**> **int**<W>: Represents the integer value using two’s complement on a large enough number of bits and keeps the least-significant *W* bits; overflow should lead to a warning
* A **struct**/**header** can be cast to another **struct**/**header** if and only if they have the same number of fields and all the fields in the source can be recursively cast to the corresponding fields in the destination.
* Given a type declaration introduced by typedef S T, values of types T and S can be cast back and forth.

### Implicit casts

Unlike C, P4 allows a very limited number of implicit casts. The reason is that often the implicit casts have a non-trivial semantics, which is invisible for the programmer.

Implicit casts are allowed in P4 only when their meaning is completely unambiguous:

* To convert an **int** value to a fixed-width type.
* In assignments (including passing arguments to method calls), when RHS has a different type from LHS.

Most binary operations that take an **int** and a fixed-width operand will insert an implicit cast to convert the **int** operand to the type of the fixed-width operand.

Consider a program with the following values:

**bit**<8> x;

**bit**<16> y;  
**int**<8> z;

The following expressions are translated by the compiler as follows:

* x + 1 becomes x + (**bit**<8>)1
* z < 0 becomes z < (**int**<8>)0
* x << 13 becomes 0; overflow warning
* x | 0xFFF becomes x | (**bit**<8>)0xFFF; overflow warning

### Illegal arithmetic expressions

Consider a program with the following values:

**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 the user is required to explicitly state it.

|  |  |  |
| --- | --- | --- |
| Expression | Why it is illegal | Alternatives |
| x + y | Different widths | (**bit**<16>)x + y  x + (**bit**<8>)y |
| x + z | Different signs | (**int**<8>)x + z  x + (**bit**<8>)z |
| (**int**<8>)y | Cannot change both size 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 | Width of 1 is unknown | 32w1 << x |
| ~1 | Bitwise operation on **int** | ~32w1 |
| 5 & -3 | Bitwise operation on **int** | 32w5 & -3 |

## Tuple expressions

Tuple expressions can only appear in in select expressions (Section 10.6). Values with tuple types are created by writing a comma-separated list of expressions enclosed within parentheses:

tupleKeysetExpression

: '(' simpleKeysetExpression ',' simpleExpressionList ')'  
 ;

For example, in the following program fragment from a parser definition we have two instances of tuple expressions:

**transition** **select**(headers.ipv4.version, ipv4.ihl) {  
 (4, 5) : process;  
 **default** : reject;  
}

The argument of the select expression is a tuple with two arguments. The label case (4, 5) is another tuple expression.

## List expressions

A list expression is similar to a tuple expression, but is introduced using curly braces instead of parentheses:

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

expressionList

: expression

| expressionList ',' expression

;

The type of a list expression is a tuple type (Section 6.2.5). List expressions can be assigned to **struct** or **header** typed-values, or they can be passed as arguments to methods. List expressions are not l-values.

For example, the following example uses a list expression to pass multiple header fields simultaneously to a non-incremental checksum unit:

**extern** ck16 {  
 **bit**<16> get<T>(**in** T data);  
}

ck16() unit; // allocate checksum unit

hdr.ipv4.cksum =   
 ck16.get( { hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,  
 hdr.ipv4.totalLen, hdr.ipv4.id, hdr.ipv4.flags,  
 hdr.ipv4.fragOff, hdr.ipv4.ttl, hdr.ipv4.proto,  
 hdr.ipv4.src, hdr.ipv4.dst } );

## Set expressions

Some P4 expressions denote *sets* of values (set<T>, for some type T; see Section 6.2.6). These expressions can appear only in limited contexts.

keysetExpression

: DEFAULT

| tupleKeysetExpression

| simpleKeysetExpression

;

tupleKeysetExpression

: '(' simpleKeysetExpression ',' simpleExpressionList ')'

;

simpleExpressionList

: simpleKeysetExpression

| simpleExpressionList ',' simpleKeysetExpression

;

simpleKeysetExpression

: expression

| expression MASK expression

| expression RANGE expression

;

The mask (&&&) and range (..) operators have the same precedence, just above &.

For example, the **select** expression (Section 10.6) has the following shape:

**select** (expression) {  
 set1 : state1;  
 set2 : state2;  
 …  
}

In this context the expressions set1, set2, etc. evaluate to sets of values. The **select** expression tests whether its argument belongs to any of the following sets.

### Singleton sets

In a set context a simple expression denotes a set containing a single element. For example:

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

4 is a set expression denoting the set consisting of the single value 4.

### The universal set

In a set context the **default** expression denotes a set containing all possible elements.

**select** (hdr.ipv4.version) {  
 4 : continue;  
 **default** : reject;  
}

### Cubes

The mask &&& infix operator takes two arguments of the same **bit**<W> type, and creates a value of type set<**bit**<W>>. The right value is a “mask”, where each 0 bit in the mask indicates a “don’t care” bit. The set denoted by a &&& b is defined as

a &&& b = { c of type bit<W> where a & b = c & b }

(This set looks like a cube in the Cartesian space {0,1}W.) For example:

8w0x0A &&& 8w0x0F

denotes a set that contains 16 different 8-bit values, whose bit-pattern is XXXX1010, where the value of an X can be any bit.

P4 targets may impose additional restrictions on the expressions on the left and right-hand side of a mask operator: for example, they may forbid non compile-time-constant expressions to be used in one or both positions.

### Ranges

The range .. infix operator takes two arguments of the same type T **bit**<W> or **int**<W> and creates a value of type set<T>. The set contains all values numerically between the first and the second, inclusively.

For example:

4w5 .. 4w8

denotes a set with values 4w5, 4w6, 4w7, and 4w8.

### Tuples

A tuple of set values is a set of tuples, e.g.:

tuple<set<T1>, set<T2>> = set<tuple<T1, T2>>

In consequence one can use tuples in a set context:

**select**(hdr.ipv4.ihl, hdr.ipv4.protocol) {  
 (4w0x5, 8w0x1): parse\_icmp;  
 (4w0x5, 8w0x6): parse\_tcp;  
 (4w0x5, 8w0x11): parse\_udp;  
 **default**: accept; }

## Operations on struct types

The only operation defined on values with a structure type is member access operation, indicated using the dot . operator (e.g., s.field). Field extraction from an l-value produces an l-value. Structs can also be copied using assignment; this is only possible between structs that have the same type.

## Operations on headers

Headers provide the same operations as structs. Assignment between headers also copies the valid header bit. In addition, headers provide an additional method setValid(bool), which sets the header’s validity.

The result of reading or writing a field in an invalid header is undefined – with the exception of the **valid** field, which can always be read. The result of reading an uninitialized header field is undefined – even if the header itself is valid.

## Expressions on header stacks

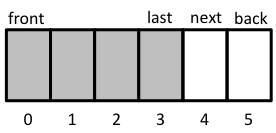


Figure 8: header stack with 6 elements

A header stack is a fixed-size array of headers with the same type. Figure 8 shows an example header stack with 6 elements, with the indices shown at the bottom. In this figure the first 4 elements are valid (shown shaded).

P4 provides a set of computations that can only be applied to header stacks. Given a value hs of type header stack, the following expressions are legal:

* hs.empty: result is a Boolean value, which is true when all valid bits in the stack are false.
* hs.full: result is a Boolean value, which is true if all valid bits in the stack are true.
* hs.next: result is a reference to the lowest-index header in the stack which does not have a valid bit set. Can only be used in a parser. Results in an assertion failure (with the standard error StackFull) if the stack is full (see section 10.7 about assertions).
* hs.last: result is a reference to the largest-index header in the stack which does have a valid bit set. Can only be used in a parser. Results in an assertion failure (with the standard error StackEmpty) if the stack is empty.
* hs[index]: result is an l-value reference to the header at the specified position within the stack. The header may be invalid. Some targets may impose the constraint that the index expression evaluates to a compile-time constant.

[TODO: We can rewrite the semantics of last and next not to assert – and then we can use them in the control pipeline as well.]

In addition, header stacks offer the following two methods that return **void** (the count argument must be a compile-time constant):

* stack.pop\_front(int count): shift “left” by count (e.g., element with index count is copied in stack at index 0). The last count elements become invalid.
* stack.push\_front(int count): shift “right” by count. The first count elements become invalid.

[TODO: perhaps these methods should be renamed to “shift” instead of pop and push]

[TODO: emphasize semantics in the presence of “holes” in the stack.]

## Function calls, method invocations

Functions can be invoked using the function call syntax.

expression   
 : ...

| expression '<' typeArgumentList '>' '(' argumentList ')'

| expression '(' argumentList ')'

argumentList

: /\* empty \*/

| nonEmptyArgList

;

nonEmptyArgList

: argument

| nonEmptyArgList ',' argument

;

argument

: expression

;

typeArgumentList

: typeRef

| typeArgumentList ',' typeRef

;

Function arguments are evaluated in order, left to right, before the function invocation takes place. The calling convention is copy-in/copy-out (Section 5.6). For generic functions the type arguments can be explicitly specified in the function call. No implicit casting is used for function arguments; the types of the arguments *must* match the parameter types exactly.

## Constructor invocations

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

* **extern** objects
* **parser**s
* **control** blocks

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 the next section. (Instantiations are similar to constant declarations.)

The syntax of a constructor invocation is similar to a function call. Constructors are evaluated entirely at compilation-time (see Section 15). In consequence, all constructor arguments must also be expressions that can be evaluated at compilation time.

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

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

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

# Constants and variable declarations

## Constants

Constant values are defined with the syntax:

constantDeclaration

: optAnnotations CONST typeRef name '=' initializer ';'

;

initializer

: expression

;

This 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 };

initializeris an expression that evaluates to a compile-time constant.

## Variables

Local variables can be declared using variable declarations:

variableDeclaration

: annotations typeRef name optInitializer ';'

| typeRef name optInitializer ';'

;

optInitializer

: /\* empty \*/

| '=' initializer

;

Variables without an initializer list are uninitialized. The only exception is the valid bit of a **header** type variable, which is initialized by default to **false**.

It is an error to read the value of a variable that has not been set. The compiler should attempt to enforce this rule statically.

## Instantiations

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

instantiation

: typeRef '(' argumentList ')' optAnnotations name ';'

;

An instantiation looks like a constructor invocation followed by a name. Instantiations are always executed at compilation-time (Section 15.1). The effect is to allocate an object with the specified name, and to bind it to the result of the constructor invocation.

The following example shows how a hypothetical counter bank can be instantiated:

**// 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(...) {  
 Counter(32w1024, CounterType.Both) ctr; // instantiation

**apply** { ... }  
}

# Statements

Statements (except the block statement) must end with a semicolon, C-style.

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, conditionals are not supported in parsers, and **switch** statements are only supported in **control** blocks. The grammar reflects these constraints. We present here the most general case, for **control** blocks.

statement

: assignmentOrMethodCallStatement

| conditionalStatement

| emptyStatement

| blockStatement

| returnStatement

| switchStatement

;

assignmentOrMethodCallStatement

: lvalue '(' argumentList ')' ';'

| lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'

| lvalue '=' expression ';'

;

In addition, parsers support a **transition** statement (Section 10.5).

(Due to limitations in the Bison parser generator we have grouped all productions starting with a l-value in the same rule.)

## Assignment

Assignment is denoted with the = sign.

An assignment evaluates the expression on the right hand side and *copies* that value into the left hand side. Derived types (e.g. structs) are copied recursively. Headers are copied, including their valid bits. Assignment is not defined for **extern** values.

### L-values

L-values are expressions that can appear on the left side of an assignment operation. These expressions are legal l-values:

lvalue

: name

| pathPrefix name

| lvalue '.' member

| lvalue '[' expression ']'

| lvalue '[' expression ':' expression ']'

;

* Identifiers of a base or derived type
* Structure/header field member access operations (using the dot notation)
* References to elements within header stacks (see Section 7.9): 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

The meaning of an assignment to a slice is the following:

e[m:l] = x

This statement sets bits m to l of e to the bit-pattern represented by x, and leaves all other bits of e unchanged. (x is implicitly cast to an m – l + 1 bitstring value).

## The empty statement

The empty statement is a no-op.

emptyStatement

: ';'   
 ;

## The block statement

A block statement is denoted by curly braces, as in C. It contains a sequence of statements and declarations, which are executed sequentially. The variables, constants and instantiations within a block statement are only visible within the block statement.

blockStatement

: '{' statOrDeclList '}'

;

statOrDeclList

: /\* empty \*/

| statOrDeclList statementOrDeclaration

;

statementOrDeclaration

: variableDeclaration

| constantDeclaration

| statement

| instantiation

;

## The return statement

The **return** statement immediately terminates the execution of the **action** or **control** that contains the **return** statement. **return** statements are not allowed within parsers.

returnStatement

: RETURN ';'

;

## The conditional statement

The conditional statement is very similar in syntax and semantics with the corresponding C if statement. The only difference is that the only acceptable type for the expression that drives the conditional in P4 is Boolean (and not integer).

conditionalStatement

: IF '(' expression ')' statement

| IF '(' expression ')' statement ELSE statement

;

## The switch statement

The **switch** can only be used within **control** blocks.

switchStatement

: SWITCH '(' expression ')' '{' switchCases '}'

;

switchCases

: /\* empty \*/

| switchCases switchCase

;

switchCase

: switchLabel ':' blockStatement

| switchLabel ':' // fall-through

;

switchLabel

: name

| DEFAULT

;

The expression within the **switch** statement is restricted to be the result of a **table**’s invocation (more details are given in Section 11.2.2).

If a switch label is not followed by a block statement, it is fall-through to the next label. However, if a block statement is present, there is no fall-through. Note, that this is different from C switch statements, where a break is needed to prevent fall-through. It is legal to have no matching label for some actions, or no **default** label.

**switch** (t.apply().action\_run) {  
 action1: // fall-through to action2:  
 action2: { ... }  
 action3: { ... } // no fall-through from action2 to action3 labels  
}

# Packet parsing in P4

This section describes the P4 constructs specific to parsing network packets.

## Parser states

A parser describes a finite state-machine (FSM) with one start state and two final states. The start state is named with the reserved keyword 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 logically outside of the parser. Figure 9 illustrates the general structure of a parser state-machine.



Figure 9: Parser FSM structure

## Parser declarations

P4 programmers are expected to provide parser declarations for all programmable parsers of a target.

parserDeclaration

: parserTypeDeclaration optCompileParameters   
 '{' parserStatefulElements parserStates '}'  
 ;

parserStates

: parserState

| parserStates parserState

;

Prior to the parser states, a parser may also contain a list of stateful element declarations (see Section 5.5.2). These are constants or instantiations (see Section 8.3) of other objects that may be used within the parser. Such objects may be instantiations of **extern** objects, or other **parser**s or **control**s that may be invoked as subroutines. (Specific architectures may forbid the instantiation of **control** blocks within a **parser**.)

parserStatefulElement

: constantDeclaration

| instantiation

;

At least one state, named start, must be present in any parser. State declarations are described below.

For an example containing a complete declaration of a parser see Section 4.3.

For a description of the optCompileParameters, used for building parameterized parsers, see Section 12.

## The Parser abstract machine

We explain the semantics of P4 parsers program using an abstract machine that manipulates a data structure named ParserModel. The abstract machine is described using pseudo-code.

The abstract model parses each packet separately and completely before accepting a new packet. 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;  
 state currentState;

onPacketArrival(packet p) {  
 ParserModel.parseError = NoError;  
 ParserModel.currentState = start;  
 execute(ParserModel.currentState);  
 }  
}

## Parser states

A parser state has a name and a body. The body of the state describes data processing performed when the parser transitions to the specified state. This data processing may consist of:

* Data extraction from packet into headers
* Invocations of method of **extern** blocks (e.g., checksum computations)
* Assertion checks for data validity
* Transition to another state

A state is declared with the following syntax:

parserState

: STATE name '{' parserStatements transitionStatement '}'

;

A parser cannot contain two states with the same name. A parser cannot define the accept and reject states; these are logically outside of the parser. A parser must define the start state.

parserStatements

: /\* empty \*/

| parserStatements parserStatement

;

parserStatement

: assignmentOrMethodCallStatement

| variableDeclaration   
 ;

The state body contains a sequence of:

* local variable declarations
* assignment statements
* method calls. These can serve multiple purposes:
  + assertion checks
  + method invocations (e.g., for extracting data out of packets),
  + invocations of other parsers.

Certain targets may place restrictions on the complexity of the expressions that can be evaluated while executing a parser.

**Semantics:** The execution of a state entails the sequential execution of the statements in the state body.

The parser state body ends with an optional **transition** statement, which transfers control to the next state.

## Transition statements

The **transition** statement has the following syntax:

transitionStatement

: /\* empty \*/

| TRANSITION stateExpression

;

stateExpression

: name ';'

| selectExpression

;

The execution of the transition statement causes stateExpression to be evaluated, and the flow of control to be transferred to the resulting state.

**Semantics:** in terms of the ParserModel the above statement is equivalent to:

ParserModel.currentState = eval(stateExpression)

For example, this statement:

**transition** accept;

will terminate the execution of the current parser transitioning to the accept state.

If the body of a state block does not end with a **transition** statement, the implied statement is

**transition** reject;

## Select expressions

A **select** expression evaluates to a state. The syntax is the following:

selectExpression

: SELECT '(' expressionList ')' '{' selectCaseList '}'

;

selectCaseList

: selectCase

| selectCaseList selectCase

;

selectCase

: keysetExpression ':' name ';'

;

If expressionList has type tuple<T>, all keysetExpression must have type set<T>, or must be **default**.

**Semantics**: the meaning of the following **select** expression:

**select**(e) {  
 ks[0] : s[0];  
 ks[1] : s[1];  
 ...  
 **default** : sd;  
}

is defined in pseudo-code as:

key = eval(e);  
for (int i=0; i < n; i++) {  
 keyset = eval(e[i]);  
 if (keyset.contains(key)) return s[i];  
}  
assert(false, NoMatch);

Keysets are evaluated in order, from top to bottom as implied by the pseudocode above; the first keyset that includes the value in the select argument provides the result state. Note that this implies that all cases after a **default** label are unreachable. This constitutes an important difference between **select** expression and switch statement in C language, where the order does not matter).

The most typical case for using **select** expressions is to compare the value of a field from a recently-extracted header against a set of constant values, as in the following example:

**header** Ipv4\_h { … **bit**<8> protocol; … }  
**struct** P { … Ipv4\_h ipv4; … }  
P headers;

**select** (headers.ipv4.protocol) {  
 8w6 : parse\_tcp;  
 8w17 : parse\_udp;  
 **default** : accept;  
}

For example, to detect TCP reserved ports (< 1024) one could write:

**select** (p.tcp.port) {  
 16w0 &&& 16w0xFC00 : well\_known\_port; // top 6 bits are zero  
 **default** : other\_port;  
}

The expression 16w0 &&& 16w0xFC00 describes the set of 16-bit values that have their top 6 bits zero.

## Assert

Assertions provide a simple form of error handling. The assert P4 function can only be invoked within a parser. The assert function is declared in the P4 core library with the following signature:

**void** assert(**in** **bool** condition, **in** **error** err);

An assertion evaluates the first argument, the condition. If the expression is **true**, the assertion causes no side-effects. If the expression is **false**, the assertion causes an immediate transition to the reject state, which causes immediate parsing termination. Also, the err expression is evaluated, and the error object parserError associated with the parser is set to the result of this evaluation.

**Semantics:** in terms of the ParserModel the above statement is equivalent to:

ParserModel.assert(bool condition, error err) {  
 if (eval(condition) == false) {  
 ParserModel.parserError = eval(err);  
 ParserModel.currentState = reject;  
 }  
}

## Data extraction from packets

The P4 core library contains the following declaration of a built-in **extern** type called packet\_in that represents incoming network packets.

**extern** packet\_in {  
 **void** extract<T>(**out** T headerLeftValue);  
 **void** extract<T>(**out** T variableSizeHeader, **in** **bit**<32> varFieldSizeBits);  
 T lookahead<T>();  
 **bit**<32> length();  
 **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: with one argument, for extracting fixed-size headers, and one with two arguments, for extracting variable-sized headers. Because these operations can cause runtime assertion 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.

**Semantics:** we describe the semantics of these operations in terms of the following abstract model of a packet data structure (pseudo-code):

packet\_in {  
 unsigned nextBitIndex;   
 byte[] data;  
 unsigned lengthInBits;

void initialize(byte[] data) {   
 data = data;  
 nextBitIndex = 0;   
 lengthInBits = data.sizeInBytes \* 8;   
 }

bit<32> length() { return lengthInBits / 8; }  
}

### packet\_in.extract – single argument

The single-argument extract method has the following P4 declaration:

**void** extract<T>(**out** T headerLeftValue);

headerLeftValue is an expression that should evaluate to a l-value (see Section 9.1.1) of type **header** with a fixed compile-time known width. If this method executes successfully, on completion the headerLeftValue is filled with data from the packet and its valid bit is set. This method may fail by executing a failed assertion (e.g., not enough bits left in packet to fill the specified header). For example, to extract an Ethernet header one can invoke:

**struct** Result { … Ethernet\_h ethernet; … }  
**parser** P(packet\_in b, **out** Result r) {  
 b.extract(r.ethernet);  
}

[TODO: should we allow extraction to metadata?]

where T is the type of the data that has to be discarded.

**Semantics:** the semantics of extract is given in terms of the following pseudo-code method of the packet class above:

[TODO: do we need to enforce that we can’t overwrite a header?]

void packet\_in.extract<T>(out T headerLeftValue) {  
 ParserModel.assert(!headerLeftValue.valid, OverwritingHeader);  
 bitsToExtract = sizeof(headerLeftValue);  
 lastBitNeeded = nextBitIndex + bitsToExtract;  
 ParserModel.assert(lengthInBits >= lastBitNeeded, PacketTooShort);  
headerLeftValue = data.extractBits(nextBitIndex, bitsToExtract);  
 headerLeftValue.valid = true;  
 nextBitIndex += bitsToExtract;  
}

### Packet.extract – two arguments

The two-argument extract method has the following declaration:

**void** extract<T>(**out** T variableSizeHeader, **in** **bit**<32> variableFieldSize);

variableSizeHeader must be a left-value representing a header that contains *exactly one* **varbit**field*.* variableFieldSize is an expression evaluating to a **bit**<32> value which indicates the number of bits to be extracted into the unique varbit field of the header (this size is not the size of the complete header, just the size of the **varbit** field).

**Semantics:** the semantics of a two-argument extract invocation is given in terms of the following pseudo-code:

void packet\_in.extract<T>(out T headerLeftValue, in bit<32> varSzInBits) {  
 ParserModel.assert(!headerLeftValue.valid, OverwritingHeader);  
 bitsToExtract = sizeOfFixedPart(headerLeftValue) + varsize;  
 lastBitNeeded = nextBitIndex + bitsToExtract;  
 ParserModel.assert(lengthInBits >= lastBitNeeded, PacketTooShort);  
headerLeftValue = data.extractBits(nextBitIndex, bitsToExtract);  
 headerLeftValue.varbitField.size = bitsToExtract;  
 headerLeftValue.valid = true;  
 nextBitIndex += bitsToExtract;  
}

The example below shows one way that IPv4 options can be extracted 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 {

...  
 Ipv4\_no\_options\_h ipv4;  
 Ipv4\_options\_h ipv4options;  
}

**error** { InvalidOptions }

**parser** Top(**packet**\_in b, **out** Parsed\_headers headers) {

...

**state** parse\_ipv4 {  
 b.extract(headers.ipv4);  
 **assert**(headers.ipv4.ihl >= 5, InvalidOptions);  
 **transition** parse\_ipv4\_options;  
 }

**state** parse\_ipv4\_options {  
 b.extract(headers.ipv4options,   
 (bit<32>)(((bit<8>)headers.ipv4.ihl – 5) \* 32));  
 **transition** **select** (headers.ipv4.protocol) { ... }

}  
}

### packet\_in.lookahead

lookahead is a method provided by the packet\_in packet abstraction that evaluates to a set of bits from the input packet without advancing the nextBitIndex pointer. Similar to extract, it asserts if there are not enough bits in the packet. One invokes lookahead 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*.

**Semantics:** in terms of the abstract model the semantics of lookahead is given by the following pseudo-code:

T packet\_in.lookahead<T>() {  
 bitsToExtract = sizeof(T);  
 lastBitNeeded = nextBitIndex + bitsToExtract;  
 ParserModel.assert(lengthInBits >= lastBitNeeded, PacketTooShort);  
T tmp= data.extractBits(nextBitIndex, bitsToExtract);  
 return tmp;  
}

Example:

**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;  
 }  
}

### Skipping bits

There are two ways to skip over bits in an input packet without assigning them to a header:

One can extract to the underscore identifier, by specifying explicitly the type of the data:

b.extract<T>(\_)

Alternatively, one can use the advance method of the packet when the number of bits to advance is known. The meaning of this method is given in pseudo-code as:

void packet\_in.advance(bit<32> bits) {  
 lastBitNeeded = nextBitIndex + bits;  
 ParserModel.assert(lengthInBits >= lastBitNeeded, PacketTooShort);  
nextBitIndex += bits;  
}

## Parsing header stacks

During parsing a header stack provides two properties: next and last. Let us consider this header stack declaration for representing the headers of a packet with at most 10 MPLS headers:

**header** Mpls\_h {  
 **bit**<20> label;  
 **bit**<3> tc;  
 **bit** bos;  
 **bit**<8> ttl;  
}

Mpls\_h[10] mpls\_vec;

In this case the expressions mpls\_vec.last and mpls\_vec.next both represent l-values of type mpls\_h inside the mpls\_vec stack. The last property returns the last value in the stack that has a valid bit set to **true**. The next property returns the first value in the stack that has a valid bit set to **false**.

Attempting to access a last element when none is available causes an assertion failure with error EmptyStack. Attempting to access a next element when the stack is full causes an assertion failure with error FullStack.

The following example shows a simplified parser for MPLS processing:

**struct** Pkthdr {  
 Ethernet\_h ethernet;  
 Mpls\_h[3] mpls\_vec;  
 // other headers omitted  
}

**parser** X(**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\_vec.next);  
 **transition** **select**(p.mpls\_vec.last.bos) {  
 0 : parse\_mpls; // This creates a loop in the FSM  
 1 : parse\_ipv4;  
 }  
 }  
 // other states omitted  
}

## Invoking 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) { ... }

**parser** caller(packet\_in packet, **out** headers h) {  
 callee() instance; // instance of callee

**state** subroutine {  
 instance.apply(packet, h.ipv4); // invoke sub-parser  
 }  
}

**Semantics**: the semantics of a subparser 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 10: Semantics of invoking a sub-parser:   
top – original program, bottom – equivalent program.

Figure 10 shows a diagram of this process.

## Unrolling parser loops

A parser processing header stacks may introduce loops in the parser graph. Since header stacks have finite maximum depth, the parser loops that extract header stacks can be statically unrolled into larger acyclic graphs.

For example, let us consider the above example extracting MPLS headers. Figure 11 shows the FSM for this parser, which contains a self-loop, and then an unrolled version of the FSM, where the mpls state has been expanded into 3 states, one for each entry of the mpls\_vec vector.

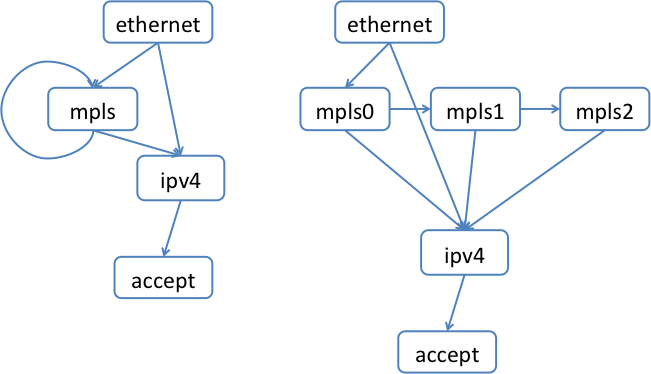


Figure 11: Unrolling a parse graph into an acyclic graph.   
The figure does not show transitions to the reject state.

A compiler may reject a parser graphs that contains a cycle which does not advance the packet\_in.nextBitIndex cursor, since it could lead to an infinite loop at runtime.

# Control blocks

P4 **parser**s are responsible for extracting bits from a packet into headers. The headers can be manipulated and transformed within **control** blocks. Control blocks are a P4 construct that can be used for describing match-action pipelines. The body of a **control** block resembles a traditional C 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 **table**s.

There is no exceptional control-flow in a **control** block: no equivalent of the assert parser statement or of the reject state. Error handling has to be performed explicitly by users.

A **control** block may start with declarations of **action**s, **table**s, constants, variables and instantiations.

controlDeclaration

: controlTypeDeclaration optCompileParameters   
 '{' stateListDeclaration APPLY controlBody '}  
 ;

stateListDeclaration

: /\* empty \*/

| stateListDeclaration stateElementDeclaration

;

stateElementDeclaration

: constantDeclaration

| actionDeclaration

| tableDeclaration

| instantiation

;

controlBody

: blockStatement

;

For a description of the optCompileParameters, which can be used to build parameterized control blocks, see Section 12.

We start by describing the core components of a **control** block, starting with actions.

## Actions

Actions are straight-line code fragments that can read and write the data being processed. In addition, actions may contain data values that can be written by the control plane and read by the data plane. In fact, actions are the fundamental construct by which the control-plane can influence dynamically the behavior of the data plane. Figure 12 shows the abstract model of an action.

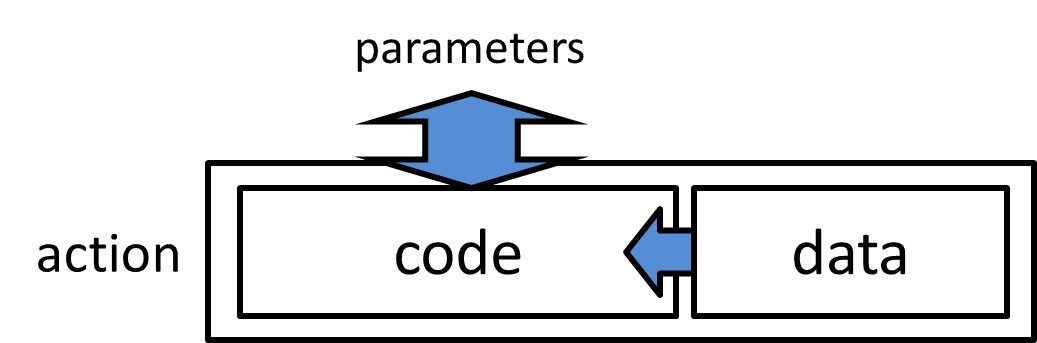


Figure 12: Actions contain code and data.   
The code is in the P4 program, while the data is set by the control plane.   
Parameters are bound by the data plane.

actionDeclaration

: ACTION name '(' parameterList ')' '{' actionElementList '}'

;

actionElementList

: /\* empty \*/

| actionElementList actionElement

;

actionElement

: assignmentOrMethodCallStatement

| variableDeclaration

| constantDeclaration

| actionBlockStatement

| emptyStatement

;

actionBlockStatement

: '{' actionElementList '}'   
 ;

Syntactically actions resemble functions with no return values. Actions may be declared within a **control** block; in this case they can only be used within an instance of that control block.

The following example shows an action declaration:

**action** Forward\_a(**out** **bit<**9> outputPort, **bit<**9> port) {  
 outputPort = port;  
}

Action parameters that have no direction (e.g., port in the previous example) indicate action data. All such parameters must be at the end of the parameter list. These parameters are bound by the control plane.

The body of an action consists is a sequence of simple statements and declarations. No conditional statements are allowed. However, simple conditional assignments can be done using the multiplexor operation (Section 7.3.1).

### Invoking actions

Actions can be executed in two ways:

* They can be invoked by tables automatically, during match-action processing
* Actions, that do not have action data (non-directional parameters), can also be explicitly invoked using function call syntax.

## Tables

The P4 **table** construct describes a match-action unit. The structure of a match-action unit is shown in Figure 13. The match-action processing consists in 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. If one desires to instantiate multiple similar tables with the same “type”, one needs to declare a table in a separate control block and instantiate that control block multiple times.

Note that a P4 table contains internally a lookup-table. This is the origin of the “**table**” name, but this overloaded use of the term “table” can be sometimes confusing. That is why we use the term “match-action unit” whenever possible in this document instead of **table**.

The look-up table is a finite map whose contents is manipulated asynchronously (read/write) by the target control-plane, through a separate control-plane API (see Figure 13).



Figure 13: Match-Action Unit Dataflow.

Syntactically a table is described by 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.

tableDeclaration

: TABLE name '(' parameterList ')' '{' tablePropertyList '}'

;

tablePropertyList

: tableProperty

| tablePropertyList tableProperty

;

tableProperty

: KEY '=' '{' keyElementList '}'

| ACTIONS '=' '{' actionList '}'

| CONST IDENTIFIER '=' initializer ';'

| IDENTIFIER '=' initializer ';'

;

The standard table properties are the following:

* 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 table.
* default\_action: an action to execute when the lookup in the lookup table fails to find a match for the key used.

We proceed discussing each of these.

A property marked as **const** has a value that cannot be changed dynamically by the control-plane. The key and actions properties are always **const**, so the specified is not needed for these.

### Table properties

#### key

The key is a **table** property which specifies the key used when looking up an action in the lookup table. The key is given by a set of pairs (expression : match\_kind):

keyElementList

: /\* empty \*/

| keyElementList keyElement

;

keyElement

: lvalue ':' name ';'

;

Each name in a keyElement must be a constant of type **match\_kind** (see Section 6.1.3).

For example, let us consider the following **table** declaration fragment:

**table** Fwd(**in** ipv4\_h ipv4header) {  
 key = {  
 ipv4header.dstAddress : ternary;  
 ipv4header.version : exact;  
 }

...  
}

In this example the lookup key is composed of two fields of the ipvheader structure: dstAddress and version. The **match\_kind** information attached to each key expression is used for two purposes:

* It is used to synthesize the control-plane API that is used to populate the table. The control-plane API specification is part of a separate document.
* It is used by the compiler back-end to allocate resources for the table’s implementation.

The P4 core library contains three predefined **match\_kind** identifiers:

**match\_kind** {  
 exact,  
 ternary,  
 lpm  
}

These identifiers correspond to the P4 V1.0 match kinds with the same names. The semantics of these annotations is actually irrelevant for describing the behavior of the P4 abstract machine; how they are used influences only the control-plane API and the actual 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 11.2.3.

If a table has no key property, then it contains no look-up table, just a default action, which is always executed (i.e., the associated lookup table is always the empty map).

#### The list of 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 attribute; the value of this attribute is always an actionList:

actionList

: actionRef ';'

| actionList actionRef ';'

;

actionRef

: optAnnotations name

| optAnnotations name '(' argumentList ')'

;

Let us consider an example from the Simple Switch program in Section 4.3:

**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 smac look-up table can contain two types of actions, named Drop\_action and Rewrite\_mac.
* The Drop\_action has an **out** argument which needs to be bound; in this case it is bound to the outCtrl.outputPort expression in the actions list.
* The Rewrite\_smac action has one parameter, but which is bound by the control plane.

#### The default action

The default action 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.

The initial value for the default action may be supplied as a value for the default\_action property. The default action may be declared as **const**, indicating that it cannot be changed dynamically by the control-plane. The default action *must* always be one of the actions that appear in the actions list.

For example, in the above **table** we could set the default action as follows (marking it also as constant – i.e., not changeable by the control plane):

**const** default\_action = Rewrite\_smac(48w0xAA\_BB\_CC\_DD\_EE\_FF);

Note that the specified default action must include all the action parameters, including the control-plane bound parameters, since this action is synthesized at compilation time.

#### Additional table properties

Tables can have additional properties. The P4 spec does not mandate any additional properties, or prescribe their semantics. Various target architectures can require attributes that are specific to these architectures.

For example, architectures where lookup-table resources are statically allocated may mandate a size table attribute, which can be used to indicate to the compiler back-end how many storage resources should be allocated.

A **table** declaration indicates the interfaces expected from a **table**: keys and actions. However, the best way to implement a table is actually dependent on the nature of the entries that will populate the table at runtime (for example, tables could be dense or sparse, could be implemented as hash-tables, associative memories, tries, etc.) An implementation attribute could also be used to pass additional information to the compiler back-end. The value of this attribute could be an instance of an **extern** block chosen from a suitable library of components. For example, the core functionality of the P4 v1.0 table “action\_profile” constructs could be implemented on target architectures that support this feature using a construct such as the following:

**extern** ActionProfile {  
 ActionProfile(bit<32> size); // how many distinct actions are expected  
}

**table** t {  
 key = { ... }  
 size = 1024;  
 implementation = ActionProfile(32); // constructor invocation  
}

(An action profile specifies the fact that, although the table is expected to have a large number of entries, only a small number of distinct entry values are expected. This can lead to an optimized implementation of the table, using an indirection level to share identical entries. The ActionProfile extern object in the previous example is meant to convey this information.)

### Invoking a table (match-action unit)

A **table** is invoked by calling its apply method. Calling an apply method on a **table** instance returns a value with a **struct** type with two fields. This structure is synthesized by the compiler automatically. For each table T, the compiler synthesizes an enum and a structure, 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;  
 action\_list(T) action\_run;  
}

The evaluation of the apply method sets the hit field to **true** if a match is found in the lookup-table. This bit can be used to drive the execution of the control-flow in the **control** block that invoked the table:

if (ipv4\_match.apply(nextHop).hit) {  
 // there was a hit  
} else {  
 // there was a miss  
}

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 **switch** statement:

**switch** (dmac.apply(nextHop).action\_run) {  
 Drop\_action: { **return**; }  
}

### Match-action unit execution semantics

The semantics of a **table** invocation statement:

m.apply( args );

is given by the following pseudo-code:

apply\_result(m) m.apply(args) {  
 apply\_result(m) result;  
  
 var lookupKey = m.buildKey(m.key, args); // 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.action\_run = action\_type(RA);  
 execute(RA);  
 return result;  
}

## 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 run-time 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.
* Applying a **table** causes the execution of the corresponding match-action unit, as described above.

# 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 optCompileParameters   
 '{' parserStatefulElements parserStates '}'

;

optCompileParameters

: /\* empty \*/

| '(' parameterList ')'

;

From this grammar fragment we infer that a **parser** declaration can have two sets of parameters:

* The runtime parser parameters
* Optional compile-time parser constructor parameters (optCompileParameters)

All compile-time parameters must be direction-less (i.e., they cannot be **in**, **out** or **inout**). When instantiating a parser one has to supply compile-time known values for all optCompileParameters.

Consider the following example:

**parser** GenericParser(**packet**\_in b, // parser API  
 **out** Packet\_header p)  
 (**bool** udpSupport) { // constructor parameters  
 EthernetParser() ethParser;

**state** start {  
 ethParser.apply(b, p.ethernet);  
 **transition** **select**(p.ethernet.etherType) {  
 16w0x0800 : ipv4;  
 }  
 }

**state** ipv4 {  
 b.extract(p.ipv4);  
 **transition** **select**(p.ipv4.protocol) {  
 6 : tryudp;  
 17 : tcp;  
 }  
 }

**state** tryudp {  
 **transition** **select**(udpSupport) {  
 **false** : accept;  
 **true** : udp;  
 }  
 }

**state** udp {  
 ...  
 }  
}

When instantiating the GenericParser one must supply a value for the udpSupport parameter, as in the following example:

// TopParser is a GenericParser where udpSupport = false  
GenericParser(**false**) TopParser;

# Packet construction (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 sequence from the Simple Switch example writes first an Ethernet header and then an IPv4 header into a packet\_out:

**control** TopDeparser(packet\_out b, **in** Parsed\_packet p) {  
 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 indexes.

## Data insertion into packets

The packet\_out datatype is defined in the P4 core library, and reproduced below. It provides two methods for appending data to an output packet; both methods are called emit. The first version only accepts headers, and the second one accepts arbitrary data.

**extern** packet\_out {  
 **void** emit<T>(**in** T hdr);  
 **void** emit<T>(**in** **bool** condition, **in** T data);  
}

The first version of emit just calls the second one with the header valid bit as a condition.

We describe the meaning of these methods in pseudo-code as follows:

packet\_out {  
 byte[] data;  
 unsigned lengthInBits;

void initializeForWriting() {  
 data.clear();  
 lengthInBits = 0;  
 }

// append entire header if it is valid  
 // T must be a header type  
 void emit<T>(T header) {

emit(header.valid, header);  
 }

// append the data to the packet if the condition is true  
 void emit<T>(bool cond, T data) {  
 if (!cond) return;  
 data.append(data);  
 lengthInBits += data.lengthInBits;  
 }  
}

We describe the two-argument emit method. For a base type T, emit:

* does nothing if the condition is false,
* otherwise it appends the data value to the tail of the packet\_out.

For derived type, emit recursively proceeds on fields:

* If the argument is a header, its valid bit is AND-ed with the condition bit to determine; if the result is **false** no action is taken
* If the argument is a header stack:
  + The emit statement is applied to each component of the stack starting from the element with index 0.
* If the argument is a struct containing multiple fields
  + The emit is recursively applied to each component of the struct in the order of their declaration in the struct.

Appending a bit-string or integer value to a packet\_out writes the value starting with the most-significant bit. This process is the inverse of data extraction.

# Architecture description

The target 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 Simple Switch declaration from Section 4.1.

The architecture description file may pre-define data types, constants, helper package implementations, and errors. It must also declare the types *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 parameters (type variables). This mechanism is similar to the use of generic types or templates in languages such as C++ and Java.

packageTypeDeclaration

: optAnnotations PACKAGE name optTypeParameters   
 '(' parameterList ')'

;

## 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<\_, IH, OH> map,  
 Deparser<OH> d);  
**package** Egress<T, IH, OH>(Parser<IH> p,Port  
 EPipe<\_, IH, OH> map,  
 Deparser<OH> d);

**package** Switch<T>( // Top-level switch contains two packages  
 // type types Ingress.IH and Egress.IH may be different  
 Ingress<T, \_, \_> ingress,  
 Egress<T, \_, \_> egress  
);



Figure 14: Switch architecture fragment implied by the previous set of declarations.

Just from these declarations we can infer some useful information about the architecture of the described switch, as shown in Figure 14:

* The switch contains two separate **package**s Ingress and Egress
* The Parser, IPipe and Deparser in the Ingress package are chained in this order. 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 one outputs a value of type T, which is an input to the second.
* Note that the Ingress type IH and the Egress type IH are independent. If we had wanted to show that they are the same type, we would have instead declared IH and OH at the switch level: **package** Switch<IH, OH, T>.

## Target program instantiation

In order to build a program for the target, the compiled P4 program has to instantiate a top-level package by passing values for all its arguments creating a variable called main in the toplevel namespace. The types of the arguments have to match the types of the parameters – after a suitable substitution of the 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 prototypes:

**parser** Prs<T>(packet\_in b, **out** T result);  
**control** Pipe<T>(**in** T data); **package** Switch<T>(Prs p, Map map);

and the following declarations:

**parser** P(packet\_in b, **out** **bit**<32> start) { … }  
**control** Pipe1(**in** **bit**<32> data) { … }  
**control** Pipe2(**in** **bit**<8> data) { … }

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:

Switch<**bit**<32>>(P(), Pipe1()) main;

## A Packet Filter Model

To illustrate the versatility of P4 architecture description language, we give an example of another target architecture, which models a packet filter that makes a drop/no drop decision based only on the computation in a P4 parser.



Figure 15: A packet filter target model.   
The parser computes a Boolean value, which   
is used to decide whether the packet is dropped.

This model could be used to program packet filters running in the Linux kernel. For example, we could replace the TCP dump language with the much more powerful P4 language; P4 can support seamlessly 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 TCP dump 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 target abstract model is a packet filter.

The declaration for this architecture is very simple:

**parser** \_parser<H>(packet\_in packet, **out** H headers);  
**control** \_filter<H>(**inout** H headers, **out** **bool** accept);

**package** Filter<H>(\_parser<H> p, \_filter<H> f);

# The P4 abstract machine: evaluating a P4 program

The evaluation of a P4 program is done in two stages:

* **static evaluation**: at compilation time the P4 program is analyzed and all stateful blocks are instantiated.
* **dynamic evaluation**: at runtime each P4 functional block is executed atomically, in isolation, when it receives control from the target architecture

## Compile-time evaluation

All stateful values are instantiated at compilation time. Evaluation proceeds in order of declarations, starting in the top-level namespace:

* All declarations (e.g., **parser**s, **control**s, types, constants) evaluate to themselves.
* 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 all be either compile-time constants, or stateful object instances. 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.

For example, let us consider the following program fragment:

// architecture declaration

**parser** p(...);  
**control** c(...);  
**control** d(...);

**package** Switch(p prs, c ctrl, d dep);

**extern** Ck16 { ... }

// user code

Ck16() ck16; // checksum unit instance

**parser** TopParser(...)(Ck16 unit) { ... }

**control** Pipe(...) { ... }

**control** TopDeparser(...)(Ck16 unit) { ... }

Switch(TopParser(ck16),  
 Pipe(),  
 TopDeparser(ck16)) main;

The evaluation of this program proceeds as follows:

1. The declarations of p, c, d, Switch and Ck16 all evaluate as themselves.
2. The Ck16() ck16 instantiation is evaluated and it produces an object named ck16 with type Ck16.
3. The declarations for TopParser, Pipe and TopDeparser evaluate as themselves.
4. The main variable instantiation is evaluated:
   1. The arguments to the constructor are evaluated recursively
   2. TopParser(ck16) is a constructor invocation
      1. Its argument is evaluated recursively; it evaluates to the ck16 object
   3. The constructor itself is evaluated, leading to the instantiation of an object of type TopParser
   4. Similarly, Pipe() and TopDeparser(ck16) are evaluated as constructor calls.
   5. All the arguments of the Switch package constructor have been evaluated (they are an instance of TopParser, and instance of Pipe, and an instance of TopDeparser). Their signatures are matched with the Switch declaration.
   6. Finally, the Switch constructor can be evaluated. The result is an instance of the Switch package (that contains inside 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 16 shows the result of the evaluation in a graphical form. The result is always a graph of instances. There is only one instance of ck16, which is 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.



Figure 16: Evaluation result.

## Dynamic evaluation

The dynamic evaluation of a P4 program is orchestrated by the target model. Each target 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 Simple Switch architecture declaration 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 (the various abstract machines). All local resources (including run-time top-level parameters, local variables, etc.) are private to the execution.

### Concurrency model

[TODO: is this concurrency model suitable?]

In practice a network device may be processing multiple packets simultaneously:

* Packets may be received concurrently on different network interfaces
* Packet processing may be pipelined, with a new packet starting before the completion of the previous one

As long as the packet processing involves stateless elements and read-only state elements there should be no difference in the results obtained from concurrent or purely sequential execution.

Since **table**s are read-only from the data-plane point of view, we can provide a very simple semantics for P4 programs written solely in the P4 core language: they should behave identically irrespective of the concurrent execution.

However, as soon as one is using any stateful **extern** constructs, the question arises with respect to the semantics of the program under concurrent execution. For example, given a set of counters that can be accessed by multiple actions, what is the interleaving of the execution of the counter methods when processing multiple packets? What is the interleaving of method invocations if the counter is accessed from different blocks (e.g., ingress and egress pipelines)?

The answer to this question is left partially to the discretion of the target architecture. An architecture could:

* Prescribe specific order
* Forbid resources that are shared between multiple blocks (e.g., each counter must be allocated in one pipeline exclusively, and it must be used only from actions that can appear within one single table)
* Prescribe an implementation-specific order

We suggest the following minimum constraints on *any* P4 implementation:

* The invocation of a **table** is atomic
* The execution of a parser **state** is atomic

# Annotations

Annotations are similar to C# attributes and Java annotations. They are a simple mechanism for extending the P4 language to some limited degree without changing the grammar. To some degree they subsume the C #pragmas.

Annotations can be added to types, fields, variables, etc. using the @ syntax (as shown explicitly in the P4 grammar):

optAnnotations

: /\* empty \*/

| annotations

;

annotations

: annotation

| annotations annotation

;

annotation

: '@' name

| '@' name '(' expression ')'  
 | '@' name '(' STRING\_LITERAL ')'

;

### Predefined annotations

We pre-define a set of “standard” annotations. We expect that this list will grow.

#### 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. They have no arguments.

* @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  
 }  
}

#### Legacy support annotations

These annotations are applied to types. They are useful for control-plane tools and protocols, for example by allowing them to use custom literals to represent values of these types. These annotations have no arguments.

* @ipv4address indicates an IPv4 address; this allows the use of quad notation and masks in the control-plane API.
* @ethernetaddress indicates a 6-byte standard Ethernet address.
* @ipv6address indicates a 128-bit IPv6 address.

### 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 the C language.

[TODO: we should reserve some annotation names for the internal compiler use]

# Appendix: P4 reserved keywords

The following table shows all P4 reserved keywords. Some identifiers are treated keywords only in specific contexts (e.g., the keyword actions).

|  |  |  |  |
| --- | --- | --- | --- |
| action | bit | bool | const |
| control | default | else | enum |
| error | extern | false | header |
| if | in | inout | int |
| package | parser | out | return |
| select | state | struct | table |
| transition | true | typedef | varbit |
| void |  |  |  |

# 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.

**struct** Version{ **bit<**8> major; **bit<**8> minor;}

**const** version P4\_LIBRARY\_VERSION = { 8w1, 8w0 };

**error** {  
 NoError,  
 PacketTooShort, // not enough bits in packet for extract  
 NoMatch, // match statement has no matches  
 EmptyStack, // reference to .last in an empty header stack  
 FullStack, // reference to .next in a full header stack  
 OverwritingHeader // one header is extracted twice  
}

**extern** packet\_in {   
 // packet abstraction  
 **void** extract<T>(**out** T hdr**)**;  
 **void** extract<T>(**out** T variableSizeHeader,   
 **in** **bit<**32> variableFieldSize);  
T lookahead<T>();  
 **void** advance(**in** **bit**<32> sizeInBits);  
 **bit**<32> length(); //packet length in bytes  
}

**extern** packet\_out {  
 **void** emit<T>(**in** T hdr);  
 **void** emit<T>(**in** **bool** condition, **in** T data);  
}

**extern** **void** assert(**in** **bool** b, **in** **error** e);

**action** NoAction() {}

**match\_type** {  
 exact,  
 ternary,  
 lpm  
}

# Appendix: checksums

There are no built-in constructs in P4 for manipulating packet checksums. We expect that all 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 for computing 16-bit one’s complement using an object with a signature such as:

**extern** Checksum16 {  
 **void** clear(); // prepare unit for computation  
 **void** update<T>(T data); // add data to 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(header.ipv4); // write header  
assert(ck16.get() == 16w0, IPv4ChecksumError); // check that checksum is 0

IP checksum generation could be done as:

header.ipv4.hdrChecksum = 16w0;  
ck16.clear();  
ck16.update(header.ipv4);  
header.ipv4.hdrChecksum = ck16.get();

# Appendix: P4 v1.2 grammar

This is the grammar of P4 v1.2 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 three kinds of identifiers:

* Type names previously introduced (TYPE tokens)
* Namespaces (NAMESPACE token)
* 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 { ... }  
t x;  
**parser** p(**bit**<8> b) { ... }

The lexer has to return the following terminal kinds:

t – TYPE  
s – TYPE  
x – IDENTIFIER  
p – NAMESPACE  
b – IDENTIFIER

This grammar has been heavily influenced by limitations of the bison parser generator tool.

Several other constant terminals appear in these rules:

* SHL is <<
* LE is <=
* GE is >=
* NE is !=
* EQ is ==
* PP is ++
* AND is &&
* OR is ||
* MASK is &&&
* RANGE ..
* DONTCARE \_

The STRING\_LITERAL token corresponds to a doubly-quoted string containing only printable ASCII characters and spaces. No escape sequences are supported.

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

;

declaration

: constantDeclaration

| externDeclaration

| actionDeclaration

| parserDeclaration

| typeDeclaration

| controlDeclaration

| instantiation

| errorDeclaration

| matchKindDeclaration

;

name

: IDENTIFIER

| APPLY

| KEY

| ACTIONS

| STATE

;

optAnnotations

: /\* empty \*/

| annotations

;

annotations

: annotation

| annotations annotation

;

annotation

: '@' name

| '@' name '(' expression ')'

| '@' name '(' STRING\_LITERAL ')'

;

parameterList

: /\* empty \*/

| nonEmptyParameterList

;

nonEmptyParameterList

: parameter

| nonEmptyParameterList ',' parameter

;

parameter

: optAnnotations direction typeRef name

;

direction

: IN

| OUT

| INOUT

| /\* empty \*/

;

packageTypeDeclaration

: optAnnotations PACKAGE name optTypeParameters   
 '(' parameterList ')'

;

instantiation

: typeRef '(' argumentList ')' optAnnotations name ';'

;

optCompileParameters

: /\* empty \*/

| '(' parameterList ')'

;

pathPrefix

: '.' relativePathPrefix

| nonEmptyRelativePathPrefix

;

relativePathPrefix

: /\* empty \*/

| relativePathPrefix NAMESPACE '.'

;

nonEmptyRelativePathPrefix

: relativePathPrefix NAMESPACE '.'

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PARSER \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

parserDeclaration

: parserTypeDeclaration optCompileParameters   
 '{' parserStatefulElements parserStates '}'

;

parserStatefulElements

: /\* empty \*/

| parserStatefulElements parserStatefulElement

;

parserStatefulElement

: constantDeclaration

| instantiation

;

parserTypeDeclaration

: optAnnotations PARSER name optTypeParameters   
 '(' parameterList ')'

;

parserStates

: parserState

| parserStates parserState

;

parserState

: STATE name '{' parserStatements transitionStatement '}'

;

parserStatements

: /\* empty \*/

| parserStatements parserStatement

;

parserStatement

: assignmentOrMethodCallStatement

| variableDeclaration

;

transitionStatement

: /\* empty \*/

| TRANSITION stateExpression

;

stateExpression

: name ';'

| selectExpression

;

selectExpression

: SELECT '(' expressionList ')' '{' selectCaseList '}'

;

selectCaseList

: selectCase

| selectCaseList selectCase

;

selectCase

: keysetExpression ':' name ';'

;

keysetExpression

: DEFAULT

| tupleKeysetExpression

| simpleKeysetExpression

;

tupleKeysetExpression

: '(' simpleKeysetExpression ',' simpleExpressionList ')'

;

simpleExpressionList

: simpleKeysetExpression

| simpleExpressionList ',' simpleKeysetExpression

;

simpleKeysetExpression

: expression

| expression MASK expression  
 | expression RANGE expression

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONTROL \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

controlDeclaration

: controlTypeDeclaration optCompileParameters   
 '{' stateListDeclaration APPLY controlBody '}'

;

controlTypeDeclaration

: optAnnotations CONTROL name optTypeParameters   
 '(' parameterList ')'

;

stateListDeclaration

: /\* empty \*/

| stateListDeclaration stateElementDeclaration

;

stateElementDeclaration

: constantDeclaration

| actionDeclaration

| tableDeclaration

| instantiation

;

controlBody

: blockStatement

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* EXTERN \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

externDeclaration

: EXTERN name optTypeParameters '{' methodPrototypes '}'

| EXTERN functionPrototype

;

methodPrototypes

: /\* empty \*/

| methodPrototypes methodPrototype

;

functionPrototype

: typeOrVoid name optTypeParameters '(' parameterList ')' ';'

;

methodPrototype

: functionPrototype

| TYPE optTypeParameters '(' parameterList ')' ';'

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TYPES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typeRef

: baseType

| typeName

| DONTCARE

| specializedType

| headerStackType

;

typeName

: TYPE

| pathPrefix TYPE

;

headerStackType

: typeName '[' expression ']'

;

specializedType

: pathPrefix TYPE '<' typeArgumentList '>'

| TYPE '<' typeArgumentList '>'

;

baseType

: BOOL

| ERROR

| BIT

| BIT '<' INTEGER '>'

| INT '<' INTEGER '>'

| VARBIT '<' INTEGER '>'

;

typeOrVoid

: typeRef

| VOID

| name // may be a type variable

;

optTypeParameters

: /\* empty \*/

| '<' typeParameterList '>'

;

typeParameterList

: name

| typeParameterList ',' name

;

typeArgumentList

: typeRef

| typeArgumentList ',' typeRef

;

typeDeclaration

: derivedTypeDeclaration

| typedefDeclaration

| parserTypeDeclaration ';'

| controlTypeDeclaration ';'

| packageTypeDeclaration ';'

;

derivedTypeDeclaration

: headerTypeDeclaration

| structTypeDeclaration

| enumDeclaration

;

headerTypeDeclaration

: optAnnotations HEADER name '{' structFieldList '}'

;

structTypeDeclaration

: optAnnotations STRUCT name '{' structFieldList '}'

;

structFieldList

: /\* empty \*/

| structFieldList structField

;

structField

: optAnnotations typeRef name ';'

| optAnnotations typeRef TYPE ';' // field named like type

;

enumDeclaration

: optAnnotations ENUM name '{' identifierList '}'

;

errorDeclaration

: ERROR '{' identifierList '}'

;

matchKindDeclaration

: MATCH\_KIND '{' identifierList '}'

;

identifierList

: name

| identifierList ',' name

;

typedefDeclaration

: TYPEDEF typeRef optAnnotations name ';'

| TYPEDEF derivedTypeDeclaration optAnnotations name ';'

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* STATEMENTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

assignmentOrMethodCallStatement

: lvalue '(' argumentList ')' ';'

| lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'

| lvalue '=' expression ';'

;

emptyStatement

: ';'

;

returnStatement

: RETURN ';'

;

conditionalStatement

: IF '(' expression ')' statement

| IF '(' expression ')' statement ELSE statement

;

statement

: assignmentOrMethodCallStatement

| conditionalStatement

| emptyStatement

| blockStatement

| returnStatement

| switchStatement

;

blockStatement

: '{' statOrDeclList '}'

;

statOrDeclList

: /\* empty \*/

| statOrDeclList statementOrDeclaration

;

switchStatement

: SWITCH '(' expression ')' '{' switchCases '}'

;

switchCases

: /\* empty \*/

| switchCases switchCase

;

switchCase

: switchLabel ':' blockStatement

| switchLabel ':'

;

switchLabel

: name

| DEFAULT

;

statementOrDeclaration

: variableDeclaration

| constantDeclaration

| statement

| instantiation

;

/\*\*\*\*\*\*\*\*\*\*\*\* TABLES \*\*\*\*\*\*\*\*\*\*\*\*\*/

tableDeclaration

: TABLE name '(' parameterList ')' '{' tablePropertyList '}'

;

tablePropertyList

: tableProperty

| tablePropertyList tableProperty

;

tableProperty

: KEY '=' '{' keyElementList '}'

| ACTIONS '=' '{' actionList '}'

| CONST IDENTIFIER '=' initializer ';'

| IDENTIFIER '=' initializer ';'

;

keyElementList

: /\* empty \*/

| keyElementList keyElement

;

keyElement

: lvalue ':' name ';'

;

actionList

: actionRef ';'

| actionList actionRef ';'

;

actionRef

: optAnnotations name

| optAnnotations name '(' argumentList ')'

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ACTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

actionDeclaration

: ACTION name '(' parameterList ')' '{' actionElementList '}'

;

actionElementList

: /\* empty \*/

| actionElementList actionElement

;

actionElement

: assignmentOrMethodCallStatement

| variableDeclaration

| constantDeclaration

| actionBlockStatement

| emptyStatement

;

actionBlockStatement

: '{' actionElementList '}'

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* VARIABLES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

variableDeclaration

: annotations typeRef name optInitializer ';'

| typeRef name optInitializer ';'

;

constantDeclaration

: optAnnotations CONST typeRef name '=' initializer ';'

;

optInitializer

: /\* empty \*/

| '=' initializer

;

initializer

: expression

;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Expressions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

argumentList

: /\* empty \*/

| nonEmptyArgList

;

nonEmptyArgList

: argument

| nonEmptyArgList ',' argument

;

argument

: expression

;

expressionList

: expression

| expressionList ',' expression

;

member

: name

| TYPE

;

lvalue

: name

| pathPrefix name

| lvalue '.' member

| lvalue '[' expression ']'

| lvalue '[' expression ':' expression ']'

;

expression

: INTEGER

| TRUE

| FALSE

| name

| pathPrefix name

| expression '[' expression ']'

| expression '[' expression ':' expression ']'

| '{' expressionList '}'

| '(' expression ')'

| '!' expression

| '~' expression

| '-' expression

| '+' expression

| typeName '.' member

| expression '.' member

| expression '\*' expression

| expression '/' expression

| expression '%' expression

| expression '+' expression

| expression '-' expression

| expression SHL expression // <<

| expression '>''>' expression // check that >> are adjacent

| expression LE expression // <=

| expression GE expression // >=

| expression '<' expression

| expression '>' expression

| expression NE expression // !=

| expression EQ expression // ==

| expression '&' expression

| expression '^' expression

| expression '|' expression

| expression PP expression // ++

| expression AND expression // &&

| expression OR expression // ||

| expression '?' expression ':' expression

| expression '<' typeArgumentList '>' '(' argumentList ')'

| expression '(' argumentList ')'

| typeRef '(' argumentList ')'

| '(' typeRef ')' expression

;