STEVE - A PROGRAMMING LANGUAGE FOR PACKET PROCESSING

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Hoang Nguyen

August, 2016
ABSTRACT

Software-defined networking (SDN) promises to make network switches programmable to enable a class of intelligent networking applications that can automate network flow direction in ways that conventional switches cannot. We present Steve, a protocol-independent, domain-specific language (DSL) for writing these networking applications on SDN devices. Steve provides high-level language features for expressing protocol structure, decoding rules, forwarding decisions, packet manipulation, and event handling for reactive non-distributed control planes. These features define a packet processing pipeline – the algorithm used to make forwarding decisions.

Steve solves two issues in SDN language development: safe packet access and safe pipeline composition. Vulnerabilities in an application running a network switch can be disastrous, therefore the Steve compiler is designed to catch potential errors. Steve uses a type and constraints system which enforces these safety guarantees. To verify our work, we produced a Steve language compiler which implements these safety guarantees. We also present four compilable Steve applications: a MAC and IPv4 learning switch, a stateless firewall, and a wire. These applications are tested with a runtime environment which provides Steve access to switch resources.
Thanks to my parents for seeing me through graduate school and encouraging me to succeed.

Thanks to all the people who helped make the Steve language possible. Thanks to Dr. Andrew Sutton for being a great advisor and providing the base compiler that Steve extended. Thanks to Michael Gruesen for developing Freeflow, the runtime environment which executes Steve programs. Thanks to Jasson Casey for developing much of the early ideas that eventually became Steve and Freeflow. Also thanks to the people at Flowgrammable, Paul Gratz, Alex Sprintson, and Luke McHale, for providing feedback throughout the development of Steve.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
</tbody>
</table>

## CHAPTER

### I. INTRODUCTION

- 1.1 Motivation: A Specialized Language for Programming SDN Switches
- 1.2 Contribution: The Steve Programming Language
- 1.3 Thesis Overview

### II. BACKGROUND RELATED WORK

- 2.1 Background
- 2.2 Prior Steve Work
- 2.3 SDN Programming Languages
- 2.4 Packet Parsers and Header Specifications
- 2.5 Software Switches

### III. THE STEVE ABSTRACT MACHINE

- 3.1 Data and Control Plane Elements
- 3.2 The Steve Pipeline
- 3.3 Application Life Cycle and Execution
- 3.4 Event Handling
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>Steve Compilation</td>
<td>99</td>
</tr>
<tr>
<td>7.4</td>
<td>Application Lifetime</td>
<td>99</td>
</tr>
<tr>
<td>VIII</td>
<td>EXPERIMENTS</td>
<td>102</td>
</tr>
<tr>
<td>8.1</td>
<td>Data Sets</td>
<td>103</td>
</tr>
<tr>
<td>8.2</td>
<td>Partial vs. Full Decodes</td>
<td>104</td>
</tr>
<tr>
<td>8.3</td>
<td>Performance of Operations</td>
<td>105</td>
</tr>
<tr>
<td>IX</td>
<td>CONCLUSION AND FUTURE WORK</td>
<td>107</td>
</tr>
<tr>
<td>9.1</td>
<td>Future Work</td>
<td>107</td>
</tr>
<tr>
<td>9.2</td>
<td>Conclusion</td>
<td>110</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>APPENDIX</td>
<td></td>
<td>116</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>An example flow table that matches on IPv4 destination and TCP protocol destination port.</td>
<td>32</td>
</tr>
<tr>
<td>7.1</td>
<td>The Freeflow ABI that may be used by applications. The ABI and its function implementations are subject to change, so specific parameters and return types are not given.</td>
<td>100</td>
</tr>
<tr>
<td>7.2</td>
<td>The application interface. All Freeflow loadable applications must provide this interface.</td>
<td>101</td>
</tr>
<tr>
<td>8.1</td>
<td>The three pcap test cases used for experiments.</td>
<td>103</td>
</tr>
<tr>
<td>8.2</td>
<td>Comparing firewall performance (in Mpps) with partial header decodes versus full header decodes.</td>
<td>104</td>
</tr>
<tr>
<td>8.3</td>
<td>Average wall clock time for executing certain operations. Output action has been excluded as it varies with the threading model implementation of Freeflow.</td>
<td>106</td>
</tr>
<tr>
<td>A.1</td>
<td>Steve reserved keywords. Note that Steve reserves the right to make any identifiers keywords in future versions.</td>
<td>118</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Openflow switch architecture is made up of ports, a pipeline, and an external controller which uses OpenFlow messages to communicate. (Image Source: [1])</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Steve program entities and how the split between control and data planes. The data plane’s pipeline raise events which are handled by event handlers on the control plane. Event handlers make system calls back to the data plane to modify pipeline behavior or to re-enter special packets.</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Packets entering the device go through ingress processing. Then a pipeline composed of decoders and tables decide where the packet should be forwarded. Once the decision is made, the packet goes through egress processing before being forwarded.</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The Steve binding environment maps field names to stacks of ((offset, length)) pairs. This allows multiple fields of the same name to be extracted and also supports fields being rebound with new names.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>A demonstration of the decoding process in action.</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Kinds of pipeline composition.</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>An ill-formed pipeline. Table T1 requires fields A, B, and C. However, in the path from D1 to D3 to T1, field C has not been extracted.</td>
<td>89</td>
</tr>
<tr>
<td>7.1</td>
<td>The FFVM architecture. FFVM virtualizes the underlying switch hardware and switch OS. Steve programs are loaded by FFVM and instantiates the data plane’s pipeline. The Steve program also serves as the device’s controller.</td>
<td>98</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

The Internet is nowadays ubiquitous with everyday life. In the last twenty years, the industry has rapidly evolved to incorporate all kinds of new Internet services: cloud services, web services, Internet of Things, etc. As the world grows more reliant on services provided over a network, so must the communication network evolve.

The backbone of all networks is the switch. The term “switch” is used in this thesis to mean any device capable of making forwarding decisions across multiple layers of the OSI stack [2]. It thus incorporates bridges, routers, hardware firewalls, etc. The problem is that modern switches suffer from a number of issues which make them difficult to use in an industry where networks are becoming increasingly elaborate.

Conventional switches are complex and unscalable because they are only designed to deal with very specific sets of well-known protocols and use cases. A router, for example, may be reconfigured with new forwarding rules, but it will never be anything other than a router. The behavior is difficult if not impossible to change, making these devices very rigid. Because they may only be reconfigured within a limited use case, they cannot easily adapt to dynamic changes in network traffic and
load. Even when reconfiguring, network administrators are often forced to do so through clunky, vendor-imposed command line interfaces.

However, future networks will require that network flows be directed in increasingly complex ways. More protocols and information will be carried on packets than ever before, and the modern switch model is not flexible enough to handle it all. In order to bypass this inflexibility, it must be possible to implement network functions (routing, firewalls, NAT, etc) through intelligent networking applications implemented in software. These applications must be capable of automatically adapting to changing network conditions by modifying switch behavior to optimize forwarding decisions, throughput, resource consumption, etc. There must also exist a switch which is capable of installing and running this software.

This is the promise of software-defined networking (SDN). SDN aims to make two elements of a conventional network switch programmable: the data plane and the control plane. The data plane, sometimes called the forwarding plane, is the element which is responsible for decoding packet content and deciding which port to forward to based on that content. The control plane manages the data plane. It is the element which learns and configures the forwarding logic used by the data plane. It is also capable of processing packets which are too complex or slow for the data plane.

The Open Network Foundation (ONF) defines SDN as the “separation of the network control plane from the forwarding [data] plane” [3]. The control plane is replaced by a software application known as a controller (software control plane), which manages switch behavior. It may be run on any machine, completely separate
from its switch and data plane. It may then perform distributed control over multiple
data planes by communicating via network messages. The most widely adopted
standard for this communication is OpenFlow [1]. Because it may be run on any
machine, any programmer can write regular software for common operating systems
and processors to control the behavior of their networking device.

The **programmable data plane** is implemented with specialized, programmable
packet processors. Software may then be installed to program this hardware. A
programmable data plane provides a key benefit that the SDN controller cannot:
*protocol independence*.

Protocol independence means that a networking device is not required, by
default, to support a suite of well-known protocols (Ethernet, IPv4, TCP, etc). Rather
than relying on specialized hardware or firmware tailored towards parsing/decoding
these protocol headers, the data plane supports processors and generic instructions
that can be used operate on any header and field. This allows users to support any
future protocol headers in addition to supporting current, well-known ones. This
makes the data plane scalable, that is, it can adapt to new headers easily.

Because the programmable data plane can flexibly deal with any field, the
programmer can change the behavior of the network switch without changing its
hardware. One day it could be a bridge and the next it could be a firewall depending
on the installed software.
1.1 Motivation: A Specialized Language for Programming SDN Switches

Some attempt to program SDN switches using C. Others attempt to produce domain-specific languages (DSLs) for SDN devices. The focus of this thesis is one such DSL, *Steve*, a language for the specification of programs that perform packet processing and network flow direction for programmable SDN switches. Steve is tailored towards expressing network applications, and therefore provides high-level language abstractions for packet processing logic. Additionally, Steve ensures a network application is safe, while still being as efficient as possible.

A network application must be safe to execute more than anything else. Network applications, especially those running on high-traffic devices, must not crash due to logical mistakes or unnoticed undefined behaviors. At best, this produces expensive network downtime; at worst, this opens the way for security vulnerabilities on the device: denial of service attacks, buffer overflows, etc.

Unfortunately, programming network devices in C can be prone to errors stemming from buffer and resource management, or logical mistakes made by the programmer. These errors can introduce vulnerabilities or impact performance. These issues are compounded by the fact C is a general purpose language not tailored towards expressing packet processing, thus the burden of safety is left to the programmer, with no compiler to double check their work. Specifically, C programs risks errors such as:
• accessing memory outside the bounds of a packet buffer or outside a constrained region of memory,

• writing bytes which exceed a constrained subset of the packet resulting in a buffer overflow (i.e. writing new bytes into a header field but exceeding the size of that field),

• not writing enough bytes while modifying a field,

• using a field in an operation that is not supported by its range of values,

• non-terminating cycles in program logic (infinite loops triggered by a single packet),

• incorrect assumptions about decoding state. That is, a programmer using a field which they have not extracted yet or that does not exist.

Programmers using C for SDN applications are also burdened with management of device resources. Specifically, a C programmer must manage:

• memory. The programmer must manage their own buffers for holding packet data and packet meta data. This opens the opportunity for accidental memory leakage.

• ports. The programmer is responsible for receiving, reading, and forwarding through ports. This process should be decoupled from the packet processing logic for modularity. Packet processing logic should be architecture independent whereas port management is a architecture dependent problem.
• vendor optimizations. Having a specialized SDN language allows the compiler to map high-level abstractions down to specific architectures and vendor provided libraries.

Steve focuses on the language features required for programming a single device. There must be a well-designed abstract machine for programming network applications on one switch before expanding to incorporate distributed control.

There are flaws in distributed control that a single device does not have to deal with. Specifically, distributed control is slow. The switch has to send messages over a network, the controller must process the message, then send more messages back to modify switch behavior. This latency is unnecessary for simple network functions. The trade-off is that it allows for control over an entire network.

A non-distributed control plane, on the other hand, has much better performance because it can exist on the same hardware as the data plane. There is little communication latency because the two elements may communicate directly through an application binary interface (ABI) – symbols and calling conventions defining communication between the two elements. This model is considerably more effective for simple network functions. The programming model for a control and data plane on the same device is basic event-driven programming (EDP).

The data plane raises events when it needs something from the control plane. The control plane catches these events with special handlers and modifies the forwarding logic accordingly. A free benefit of this model is that a single language like Steve may be used to define both control and data plane elements for a given device.
1.2 Contribution: The Steve Programming Language

The Steve programming language was designed to solve the problem of safe and efficient programming of SDN devices. Steve is used for writing applications that define packet processing and network flow direction on SDN devices. It is protocol independent, strongly-typed, and declarative.

This thesis provides Steve language features for defining packet processing pipelines, the core component used by a data plane to process packet content and make forwarding decisions. It also provides features for defining event handlers – control plane functions which process unexpected packets and modify the data plane when necessary.

1.2.1 Language Features

The packet processing pipeline is a data plane algorithm that uses multiple, smaller processing stages to decode packet content and make forwarding decisions. The Steve pipeline is a generalization of the pipeline model defined by OpenFlow [1]. Steve provides high-level language features for specifying stages in this pipeline and how these stages connect. Specifically, this thesis describes the following contributions towards language features:

**Header Structures.** A programmer may define the structure of any protocol header using an abstract mechanism known as a layout. A layout specifies what fields are in a header, the lengths of those fields, and types for those fields.
**Decoders.** Decoders are pipeline stages used to extract fields from a packet header. They conform to user-defined layouts, making them flexible enough to handle any protocol header.

**Flow tables.** Flow tables are pipeline stages responsible for the majority of forwarding logic. A *flow* is a group of packets from a source to one or more destinations. In order to direct these flows, Steve uses *flow tables* (an OpenFlow inspired abstraction). A flow table is a dynamic decision table which classifies packets into groups (flows) using decision rules (known as *flow entries*). Packets which are part of the same flow have a common set of *actions* applied to them.

**Actions.** Actions provide a way to modify a packet’s fields, add/remove flow entries from tables, and forward/drop packets. Steve actions are a generalization of OpenFlow’s instructions and actions.

**Pipeline Composition.** A Steve pipeline is a composition of two kinds of stages: decoders and flow tables. Steve provides languages features for describing how these stages are linked together. It also provides safety guarantees on the pipeline.

**Event Handlers.** The pipeline may raise events to the control plane when it cannot handle a packet. Steve allows for the definition of *event handlers* – control plane functions which deal with these special circumstances. Unlike the pipeline, event handlers can execute a wider range of operations that are too slow or complex for a data plane such as: logging, flow table modification, C library calls, etc.
1.2.2 Language Safety

This thesis also describes solutions for ensuring language safety. Steve is a statically-typed language. The compiler performs additional work to enforce strict safety guarantees so that runtime checks can be avoided as much as possible.

The Steve type system ensures that operations on header fields are valid and well-defined. To reduce errors related to working with fields as byte buffers, Steve allows for the representation of fields as fixed-width, signed or unsigned integers. Implicit integer conversions can be applied to these fields to ensure that the correct number of bytes is always being written into the packet, making buffer overflows impossible. Additionally, Steve’s layouts prevent the programmer from making common mistakes while extracting fields or manipulating protocol headers.

The Steve compiler ensures certain guarantees about the correctness of a Steve pipeline through a specialized constraints system. Steve pipelines may be represented as directed graphs. The Steve compiler performs analysis on this graph to enforce the following constraints.

1. Fields that have not been extracted by a decoder cannot be used.

2. The traversal of a packet through a pipeline is always a forward progression.

That is, a packet may never enter a non-terminating cycle of decoders and tables.
Additionally, Steve is resource safe, that is, it never requires the user to heap allocate resources or manage device resources. All memory allocation in Steve is done on the stack, ensuring Steve applications are free of memory leakage and faster.

Steve relies on its runtime environment, Freeflow [4], to manage system resources such as ports, packet reading, and packet buffer allocations. This decouples the logic of system resource management from the packet processing logic.

1.2.3 Compiler

The Steve compiler implements these safety guarantees as well as ensuring efficient code generation for network applications. It generates LLVM intermediate representation (IR) code [5] from Steve syntax. LLVM was chosen because it is architecture-independent, extensible, and well-supported. The LLVM IR optimizing compiler is used to optimize Steve code as much as possible. The LLVM compiler infrastructure can also be modified to support special intrinsics which can be optimized for specific switch architectures. Right now, it targets an x86 architecture.

In order to validate that our language abstractions work, the Steve compiler builds modules (dynamic link libraries) loaded by the Freeflow virtual machine (FFVM). FFVM is a software data plane which also provides the Steve runtime environment [4]. FFVM abstracts underlying switch hardware and services and exposes them through an interoperable ABI which may be called by the Steve application. A Steve module provides the FFVM’s packet processing pipeline with flow table configuration, flow table entries, and packet decoders. The module also acts as the controller.
for FFVM’s data plane and therefore provides an event handler interface which the
data plane can call when it needs the control plane to process a packet.

1.3 Thesis Overview

This thesis is organized as follows. Chapter 2 describes similar works in the field of
SDN and SDN programming languages. It also provides a background into SDN and
the different approaches Steve takes to other languages of its kind.

Chapter 3 describes the Steve architecture and abstract machine. It explains
the abstract model for packet processing pipelines and event handling.

Chapter 4 demonstrates how to use Steve to write some simple network ap-
plications. These same samples are presented again in Appendix A.13. Chapter 5
describes all of the language features contributed by this thesis and their use cases.

Chapter 6 elaborates on pipeline safety guarantees, what algorithms enforce
them, and the proofs for these algorithms. It also provides explanations about other
limitations of the language and why they exist.

Chapter 7 describes the interface which must be exposed by the switch for
Steve applications to run. To validate this approach, this chapter also summarizes
Freeflow, a virtual data plane which exposes this interface to Steve [4]. It also de-
scribes the interface which a Steve application exposes to the switch.

Chapter 8 provides experiments using sample Steve applications. Packet
capture (pcap) files are run through Steve applications to measure the performance
of applications on a Linux machine. Chapter 9 provides discussion, details future
work for Steve, and provides concluding remarks about the project.

Appendix A serves as a reference manual which includes semantic, grammar,
and typing rules for the language. References to this appendix are made throughout
this thesis as needed.
CHAPTER II

BACKGROUND RELATED WORK

2.1 Background

Much of the work on the development of technologies for SDN is targeted at or inspired by OpenFlow: a specification describing the general architecture of SDN switches and the protocol that allows them to be remotely observed and controlled [1]. The key parts of the OpenFlow switch architecture, depicted in Figure 2.1, are a packet processing pipeline, an external software controller which sends and receives OpenFlow messages, and ports.

![Figure 2.1: The Openflow switch architecture is made up of ports, a pipeline, and an external controller which uses OpenFlow messages to communicate. (Image Source: [1])](image-url)
This is the architecture that most SDN researchers and vendors have focused on. Though OpenFlow is widely accepted and has driven the majority of SDN research, it has its flaws. OpenFlow is protocol dependent and unscalable, a compromise which resulted from targeting non-programmable switch processors. However, modern programmable switch processors have rapidly caught up in performance [6], making arbitrary packet processing more realizable. However, with OpenFlow, protocol independent research like Steve becomes artificially constrained.

Research into Steve used OpenFlow as a guidepost for what an abstract packet processing machine might look like and what language features may be necessary to support such a machine. However, by no means did this research strictly adhere to OpenFlow principles or semantics.

The primary concept Steve takes from OpenFlow is the pipeline processing model. An OpenFlow pipeline is defined as a sequence of one or more flow tables and a group table which handle packet lookups, decision making, and forwarding.

The Steve pipeline has two key differences. The first difference is that Steve’s pipeline is protocol independent, whereas OpenFlow specifies a limited set of supported protocol fields known as OXM fields. Because Steve is protocol independent, its pipeline model must contain a series of programmable decoders to extract fields from user-specified header structures. OpenFlow does not mention decoding because protocol dependent decoding is typically built into the switch.

The second difference is the actions used to modify the packet. Steve flattens the instruction+action design of OpenFlow into just actions. Steve actions are generic
and may work with any protocol field, whereas OpenFlow only supports specific fields. It also provides action extensions to manipulate flow tables which OpenFlow does not. Steve also does away with OpenFlow’s strict action set limitation, allowing any number of actions to be applied to a packet.

Since Steve focuses on programming a single switch that does not require distributed control, it does away with OpenFlow external controllers. Though the OpenFlow model provides the benefit of supporting distributed control, sending messages over a network is slow. Instead, Steve chooses to leverage the power of the control plane which resides on the same switch as the data plane.

2.2 Prior Steve Work

Due to the sensitive nature of network applications, the program must be provably safe to execute. Early development of Steve focused on the safe interpretation of packet memory as local objects. Casey, et al., described the semantic constraints, structural constraints (including buffer and view abstractions used by decoders), and safe access properties needed when working with packet headers as well as providing the early Steve language. The Arbiter Framework was another language implementation that checked the correctness of data paths (pipelines) and ensured that they did not violate the capabilities of switches they ran on [7].
2.3 SDN Programming Languages

Many other projects have tackled the idea of an SDN programming language. These languages may be further categorized into languages that program or configure data plane elements, and languages which program the control plane or controller.

Languages which target the data plane all have the common idea of a pipeline. These languages typically provide syntax for expressing packet decoding rules, table configuration, and packet manipulation. Controller languages tend to focus more on high-level, distributed control of networks, designing network topology, and setting up forwarding policies for an entire network.

Steve is unique in that it provides features for programming the data plane and defines the reactive elements (i.e. event handlers) of the controller. However, since Steve does not work with distributed control, it shares very little similarity with controller programming languages.

2.3.1 SDN Data Plane Programming and Configuration Languages

P4 is the most popular protocol independent language for pipeline specification [8, 9, 10] on specialized ASIC switches. P4 specifies a pipeline of programmable parsers (equivalent to Steve decoders) followed by a series of match+action tables (equivalent to flow tables). Parsing state is stored in a data structure called a “parsed representation.”

Protocol Oblivious Forwarding (POF) [11, 12, 13] provides a an assembly-like instruction set for POF SDN switches. The POF pipeline is a series of match tables
which are also responsible for decoding the fields they match on. The POF program-
ming model uses metadata as a “scratch pad” for decoding fields, requiring that the
programmer represent fields as generic \( \{ \text{offset}, \text{length} \} \) pairs, known as “search keys”.
NetASM is an intermediate representation language for programmable data planes
[14]. It aims to solve the same issues as POF, but also provides a language that is
target/device independent.

Steve is an approach between P4 and POF, but also attempts to fill in their
weaknesses. Steve’s pipeline specification syntax is similar to P4. P4’s high-level,
abstract syntax is easy to write and understand. In contrast, POF’s instruction
set (POF-FIS) [11] is too concrete, difficult to comprehend, and has no additional
language abstractions to ensure program safety. POF search keys are particularly
error prone because its not easy for a programmer count field offsets manually.

However, Steve’s abstract pipeline and decoding model is closer to POF’s.
Steve decoders are placed between tables so fields are only decoded as needed. This
is similar to how POF tables extract their own fields. Similarly, Steve uses POF’s
search key format for representing extracted fields, but the compiler generates the
keys rather than the programmer. The benefit of POF’s decoding model is granular
control over which fields from a header get extracted. P4, on the other hand, extracts
all header fields up-front, which wastes valuable processing time if those fields are not
needed. Also unlike P4, Steve allows the user to specify the complete pipeline logic,
including pipeline composition, flow entry definitions, and flow entry insertion/re-
moval. Because of this, the Steve compiler is able to reason about the safety of all

17
flow entries which could ever exist and all potential execution paths that may exist at runtime.

2.3.2 Data Plane Programming Libraries

Data Plane Development Kit (DPDK) by Intel are software libraries in C for writing packet processing applications [15]. It improves packet processing speeds on Intel processors and NICs. DPDK provides a single platform (Intel processors) for performing all packet processing tasks, thus eliminating the need for specialized hardware.

Open Data Plane (ODP) is an open source API in C for developing data plane applications [16]. ODP provides application portability by providing a common set of APIs across multiple platforms and instruction set architectures.

2.3.3 SDN Controller Programming Languages

The Frenetic project has produced a family of network programming languages. Frenetic [17, 18] and Pyretic [19] are sister languages that use SQL-like queries to classify packets and support a library for describing packet forwarding policies over a collection of network switches. The goal is to abstract away the difficulties of programming a centralized SDN controller. NetCore is similarly a language for generating classifiers from those policies [20]. NetKAT is similar to NetCore except it uses Kleene algebra [21, 22] to prove the correctness of its policies.
2.4 Packet Parsers and Header Specifications

Other early DSLs focused on binary header specification languages which inspired Steve header specification syntax [23, 24, 25]. Gibb, et al. described the design principles for parsing fields using header specifications [26]. Steve decoders may be described as a non-streaming programmable parsers according to Gibb, et al. Some other examples of programmable packet parsers are Kangaroo [27] and Berkley Packet Filter (BPF) [28]. BPF also provides a language for filtering packets. It is possible for Steve programs to similarly act as a packet filter language rather than a packet forwarding language.

2.5 Software Switches

There exist a number of software switches which can be used to emulate or abstract SDN switch hardware. Freeflow is an SDN software switch architecture developed by Flowgrammable [4]. The Freeflow virtual machine (FFVM) provides a programmable, protocol oblivious, software data plane that supports multi-tenancy. Steve is designed to generate code targeting the FFVM runtime environment. A Freeflow data plane loads Steve applications that act as its control plane and programs its packet processing pipeline. P4 can similarly target a software switch called behavioral-model v2 (bmv2) [29]. Open vSwitch (OVS) is another SDN virtual, software switch which supports programmatic extensions [30, 31, 32] and reconfiguration. Open vSwitch was designed to work in virtual environments and runs on the hypervisor.
CHAPTER III
THE STEVE ABSTRACT MACHINE

An abstract machine describes the memory model, object model, program execution semantics, operations, the types of objects be operated on, the behavior of operations, and required resources. The Steve abstract machine operates over discretized streams of data (packets) which are received and sent on ports. The abstract machine loads Steve applications that define its forwarding decisions. This chapter describes the Steve abstract machine. Specifically, this chapter describes:

1. how a Steve application divides between data and control planes,
2. application life cycle,
3. what operations and actions must be supported for packets, and
4. the packet processing pipeline which handles directing packet flow.

3.1 Data and Control Plane Elements

The Steve programming language defines packet structure, decoders, flow tables, and event handlers. These entities may be split into two categories: entities which define data plane processing and entities which define control plane logic. Figure 3.1 demonstrates this divide.
The data plane’s pipeline raise events which are handled by event handlers on the control plane. Event handlers make system calls back to the data plane to modify pipeline behavior or to re-enter special packets.

This model is unconventional, as it blurs the separation between control plane and data plane. Because Steve targets SDN switches that do not require distributed control, and support control and data plane programmability, a single Steve application can provide entities for both planes. The advantage is that a single language can express the behavior of all elements in the switch.

The packet processing pipeline is part of the abstract data plane. This is the algorithm used to process the majority of packets and make forwarding decisions. This will sometimes be referred to as the fast path because data plane processing is optimized to be simple, fast, and does not incur the overhead of control plane processing. A data plane must be able to support the pipeline model: programmable
The control plane defines control and logic over the data plane. Event handlers are functions that provide an interface for the data plane to interact with the control plane. The control plane uses these event handlers to process exceptional events raised by the pipeline. The control plane is capable of more complex computation than the data plane. When the data plane cannot handle a packet because doing so would be too complicated or too slow, it requests that the control plane handle the packet. Event handlers are further discussed in Section 3.4. In order to support Steve, a switch must support a non-distributed control plane capable of registering events and executing event handlers. Additionally, it must support function calls, automatic storage for local variables, integer arithmetic, and branching.

3.2 The Steve Pipeline

A pipeline may described as a series of stages, where each stage performs a small amount of work on a packet, and then passes that packet on to the next stage. Stages are connected in such a way that the output of one stage becomes the input to the next. Figure 3.2 depicts the abstract pipeline model.

Packet processing can be described in terms of three basic stages: 1) ingress processing, 2) pipeline processing, and 3) egress processing. The pipeline processing stage can be further decomposed into two kinds of stages: decoders and flow tables. Additionally, all packets must reach a final state known as termination, where all
Figure 3.2: Packets entering the device go through ingress processing. Then a pipeline composed of decoders and tables decide where the packet should be forwarded. Once the decision is made, the packet goes through egress processing before being forwarded.

Processing for a packet has completed. Processing is terminated after egress, but certain actions may terminate processing earlier (see 3.2.10). Termination releases all resources used to store a packet and all metadata associated with that packet.

3.2.1 Ingress Processing

Ingress processing is the first phase every packet will go through. A packet enters a network device on a port, known as its ingress port. At this time, the data plane builds a data structure known as the context around the packet. Contexts save additional data about a packet as it moves through the pipeline. They serve as the input and output between pipeline stages. Further explanation of the context can be found in Section 3.2.2.
After ingress processing, the data plane dispatches the context (which includes the packet) to the first decoding stage. The ingress processing stage is not something that is written in a Steve application. It is implicitly handled by the runtime system which manages data plane resources. One such runtime implementation is described in Chapter 7.

3.2.2 Contexts

A context is a data structure used to save and carry additional metadata and stateful information about a packet between pipeline stages. Pipeline stages may use this data structure to memoize things they have learned about a packet for future stages to use. Because pipeline stages do not store any stateful information, they use this data structure to learn contextual information about the packet. Specifically, the context data structure contains:

- The packet itself. From now on, anytime mention of the context is made, it is implied that this also includes the packet.
- The packet’s logical and physical ingress ports.
- The egress port. This field is written to during pipeline processing. It ultimately determines which port the packet gets forwarded on.
- The packet’s length.
- A tunnel identifier used for handling virtual networks.
• A *binding environment* used to memoize the offset and length of header fields. This provides pipeline stages with information about decoding state.

• A *binding environment* used to memoize the offset of protocol headers.

• An *action list* that may be written to and is executed during egress processing.

The two important data structures contained within a context are the *action list* and the *binding environment*. Actions, which are described in Section 3.2.10, are appended to the action list using a write action. These actions are executed during egress processing in the order that they were appended.

A binding environment is a data structure which maps object names to corresponding storage locations at runtime [33]. The storage location of fields is represented using an *(offset, length)* pair, where *offset* is the absolute offset of the field within the packet. These mappings are known as *bindings*. Figure 3.3a depicts the abstract structure of a binding environment while Figure 3.3b depicts the contents of a binding environment during the decoding of an IP-in-IP packet.

Since any given packet can contain one or more of any field or header with the same name, a binding environment maintains a stack of *(offset, length)* pair for every field and header. These stacks are called *binding stacks*. A binding environment is thus a mapping of field names to binding stacks. When the value of a field is needed, the topmost binding on the binding stack, i.e. the most recently extracted field with that name, shall be used.
The context binding environment.
Fields are mapped to binding stacks which store the \((offset, length)\) pair representing every extracted instance of that field.

Figure 3.3: The Steve binding environment maps field names to stacks of \((offset, length)\) pairs. This allows multiple fields of the same name to be extracted and also supports fields being rebound with new names.

The field binding environment allows for the full memoization of all extractions made during pipeline processing. It also allows fields to be rebound with new names where necessary. It is of note that this approach is heavyweight, that is, it is space consuming and may be more information than is necessary for certain network functions.

3.2.3 Decoder Stages

A packet is a chunk of raw uninterpreted binary data. In order to make meaningful decisions, the program needs to recover protocol headers whose fields contain the
control information needed to forward the packet. Decoders are functions which
operate over context objects. They are responsible for determining which groups of
bits are a header, and which subsets of that header form fields. Decoders extract
these fields, allowing them to be reinterpreted as local objects with a specified type
within the decoder or by later stages of the pipeline. Since a packet is composed of a
sequence of headers, those headers and fields may be extracted by calling a series of
decoders.

A decoder operates on exactly one header. It knows where fields are in that
header by conforming to a header layout. Layouts allow Steve programmers to specify
the structure of a protocol header: what fields it contains, the length of those fields,
and the order of those fields. The layout used by a decoder is known as its layout
rule.

Extracted fields are stored as \((\text{offset, length})\) pairs in the context binding en-
vironment. This format of decoding was inspired by work done in Protocol-Oblivious
Forwarding (POF) [12, 11, 13]. However, Steve does not require the programmer
to specify these pairs manually. The pairs are compiler generated using information
from a decoder’s layout rule.

3.2.4 Enforcing Safe Extractions Through Views

There are safety issues associated with reinterpreting foreign packet memory into local
objects within the Steve program described in prior Steve work [34]. Specifically, a
program risks reading outside the bounds of the packet, or inadvertently reading
outside the bounds of a header. To prevent this, Steve ensures that decoders only have constrained access to a subset of contiguous bytes in a packet buffer. This is known as a decoder’s view of the packet.

A decoder’s view begins where the header it decodes begins, and ends (implicitly) where that header ends. This constrained region is enforced by the decoder’s layout rule. That is, a decoder only knows about bytes up to and including the final field given in a layout and cannot know about or access any fields past that point.

A packet’s context must maintain the beginning of the current view using a view index. Once a decoder is finished executing, it is responsible for shifting this view index so that it refers to the first byte of the next header.

If at any point a decoder tries to access memory outside the bounds of its view or the packet, the packet is automatically dropped before extraction occurs.

3.2.5 Decoder Operations

Figure 3.4 demonstrates how decoders extract fields and how this view mechanism is applied on an actual packet instance. Here, the Steve application decodes a typical packet containing Ethernet, IPv4, and UDP headers.

The view index always starts at the beginning of the packet. In Figure 3.4a, the first header is Ethernet. In Figure 3.4b, the three fields of the Ethernet header are extracted.

When the compiler generates (offset, length) pairs, the offset value is relative to the view index. The absolute offset of the field is calculated using view index
(a) The view of the first (Ethernet) decoder. The beginning of the view is the beginning of the Ethernet header, which is also the beginning of the packet.

(c) When a decoder is finished working, it shifts the view to the next header. The shift moves the view index by the length of the current header – 14 bytes.

(b) The beginning of a field is discovered by its relative offset from the beginning of the view. The end is determined by the field’s length. The dst field has an offset of 0 and a length of 48 bits, src has an offset of 48 bits and a length of 48 bits, and type has an offset of 96 bits and a length of 16 bits.

Figure 3.4: A demonstration of the decoding process in action.
plus the relative offset. The absolute offset and length are saved in the context binding environment. In Figure 3.4b, the pairs (0, 48) (\texttt{ethernet.dst}), (48, 48) (\texttt{ethernet.src}), and (96, 16) (\texttt{ethernet.type}) are stored in the context.

When a decoder finishes, it shifts the view index, as seen in Figure 3.4c. The beginning of the view is moved by the length of header. The view now starts one byte after the previous header’s final byte. All decoders are responsible for shifting the view in preparation for the next decoder. Once the next decoder is reached, its view is already on the header it decodes. This process continues until all decoders are finished.

3.2.6 Byte Ordering

Steve applications currently make the assumption that all fields are in network byte order. Explicit expression of byte order for a field is future work. It automatically converts fields from network byte order to the native order of the target machine. All operations on packet fields should keep this in mind. If the field was not in network byte order, this may actually produce unexpected results.

3.2.7 Partial and Full Decodes

Steve provides full control over which fields get decoded from a header. The entire header does not have to be decoded. Steve allows for the extraction of only a subset of fields from a header. This is known as a \textit{partial decode}. In some packet processing languages like P4 [8, 9], all relevant headers have all their fields extracted before any pipeline processing is done. This is known as a \textit{full decode}. The issue is that full
decodes are inherently wasteful. Only certain fields and headers within a packet are ever really needed for a forwarding decision. Extracting any other fields is a waste of valuable processing time. Due to the way Steve memoizes extracted fields, it also wastes space. Efficiency is important when trying to achieve software processing rates comparable to hardware line rate (10Gbps to 40 Gbps).

Not all headers need to be decoded either. For example, if a networking application only needs to forward using MAC addresses, there is no reason to waste time extracting fields from IPv4 or IPv6 headers, and so on. It is up to the programmer to express how deep they wish to look into a packet.

That being said, Steve supports the full decode as well. It may be desirable to some programmers to do this in certain scenarios, thus the language does not favor one paradigm over another.

3.2.8 Flow Table Stages

Tables stages are used by the pipeline to make decisions about a packet based on its content. A table stage is a function which operates over a context. It retrieves data from the context and performs lookup against an object known as a flow table. A network flow is a series of packets from a source to a set of one or more destinations. Flow tables are used to direct network flows [1] based on header content. Flow tables are modeled as decision tables which classify packets into groups (flows) based on field values found in their headers. A common set of actions, described in Section
3.2.10, are then applied to packets which are part of the same flow. Table 3.1 provides an example flow table.

Table 3.1: An example flow table that matches on IPv4 destination and TCP protocol destination port.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Key: IP Destination</th>
<th>Key: TCP Destination port</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Match Fields</td>
</tr>
<tr>
<td>1</td>
<td>10.1.1.0</td>
<td>80</td>
<td>Set IP Dest. equal to 192.168.1.30. Output to port 1.</td>
</tr>
<tr>
<td>1</td>
<td>192.168.1.31</td>
<td>23</td>
<td>Output port 2.</td>
</tr>
<tr>
<td>0</td>
<td>Miss case</td>
<td></td>
<td>Drop packet.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

A flow table has a *key* and a set of *flow entries*. The type of a flow table is determined by its match pattern and its key. A key specifies a set of typed fields from a packet, called *key fields*, that a table matches against (or classifies with). They are the equivalent of decision attributes. Table 3.1 has two key fields: IP destination and TCP destination port.

A flow entry is equivalent to a rule in a decision table. Each flow entry is composed of *match fields*, a *priority*, an *action sequence*, and miscellaneous additional metadata (called *properties*). Table 3.1 has three sample flow entries (with properties omitted).

For each key field, each flow entry has a corresponding value for that key field, known as a *match field*. Each flow entry in the table must be uniquely identifiable by its match fields and its priority. Priority is an extra field used to unambiguously
execute exactly one flow entry if a packet matches multiple flow entries. The highest priority flow entry is always executed. In this example, the first flow entry’s match fields are 10.10.1.0 (IP destination) and 80 (TCP port destination), and its priority is 1.

When a packet is being matched against a table, the fields given by the table’s key are extracted from the context. These extracted fields, together, are called the packet key or query key. Each match field is then compared against the corresponding packet field. If all fields match, then the flow entry’s actions are executed on the packet. Actions may modify the packet, context, or flow tables and are discussed in Section 3.2.10. In this example, if a packet with an IP destination of 10.10.1.0 and a TCP port destination of 80 were matched against Table 3.1, it would match the first flow entry. It would have its IP destination changed to 192.168.1.30, then it would be forwarded to port 1.

If no flow entry matches the packet’s field values, the miss case flow entry is applied to the packet instead. By default, a miss case will drop the packet, but the behavior can be user defined. Miss cases always have the lowest possible priority amongst flow entries and each match field can be considered a wildcard.

Flow entries must be able to carry additional data (or properties). For example, the timeout property indicates how long a flow entry may remain in a flow table. Properties are current predefined within the language, but the abstract machine should be able to support arbitrary properties.
3.2.9 Table Match Patterns

There can be certain match patterns supported by a table: exact, prefix, and wildcard. Steve currently only handles the exact match pattern, but plans to extend in the future. The exact match pattern means the value of the packet field must be exactly equal to the value of the match field. A logical-XOR of the two bit-patterns should produce a result equivalent to 0.

Prefix match tables use the longest prefix match algorithm to determine matches over a bit pattern. Wildcard match tables allow for arbitrary “don’t-care” bits in a bit pattern.

3.2.10 Actions

Since data planes generally have limited processing power, they are only expected to support a limited common set of actions. Actions may modify a packet, forward it, move it to a new stage, or modify a flow table. Actions are primarily applied by flow tables, but may also appear in decoders and event handlers. All actions implicitly operate on the current context object and packet.

**Output <p>**. This forwards a copy of the packet to port p.

**Drop**. This terminates packet processing on a packet and releases packet and context memory. Egress processing is skipped.

**Decode <d>**. This action is a *transition action*. It transitions the context to another decoder. That is, it executes the decoder d on the context.
**Goto** `<t>`. This transition action sends a context from the current stage to the flow table `t`.

**Raise** `<e>`. This action sends a copy of the context to a control plane event handler `e`. The original context continues through pipeline processing as usual.

**Insert flow entry** `<match fields> <properties> <action>+ <t>`.

This inserts a flow entry into flow table `t` with the given match fields, optional properties, and action sequence. The type and number of match fields must correspond to those of `t`'s key fields. Properties include priority, timeout, and other additional information. The representation of the flow entry action sequence (`<action>+`) is implementation specific. Actions may be stored as a list of objects representing each action that get interpreted by a separate execution engine. Alternatively, the action sequence may be implemented as a function which operates over a context and emulates the behavior of applying those actions. Inserting a flow entry whose match fields conflict with an existing entry will cause the old one to be removed and replaced by the new one.

**Remove flow entry** `<match fields> <t>`.

This removes a flow entry with the given match fields from `t`. If the entry does not exist, nothing is done.

**Set** `<f> <v>`. This is used to overwrite a field, `f`, in the packet with bytes given by `v`. This action enforces that the same number of bytes being written matches the size of a field to prevent buffer overflowing. The Steve application is responsible for providing bytes in the correct byte ordering.
**Write Action `<action>`.** This allows actions to be appended to the context’s action list. Only set and output actions may be written. Actions appended to the list are executed during egress processing and may have different behavior when executed. Specifically, a written output action modifies the context’s egress port field rather than forwarding a copy of the packet.

**Clear.** The clear action removes all actions from the action list.

### 3.2.11 Pipeline Composition

Kinds of processing stages can be interleaved together in any order within the pipeline unlike other languages like P4 [10, 8]. Upon entering the pipeline, a packet must first go through at least one decoding stage before moving to the next processing stage. After that, there are many possibilities. Steve pipelines allow a user to specify the following pipeline compositions.

1. **A complete decode of the packet followed by a sequence of tables.**

   Packets have all necessary headers and fields decoded and saved in the context first. The packet is then dispatched to the first table in the pipeline which will send it to more tables until a forwarding decision is made. Figure 3.5a depicts this pipeline.

2. **A chain of partial decodes and table lookups.** Packets get partially decoded and dispatched to a table. The flow entry could carry the packet to another table, another decoder, or forward it out of the network. The pipeline
is thus a chain of alternating sequences of decoding stages and table matching stages. Figure 3.5b depicts this pipeline.

3. **Only decodes.** In some extreme cases, it may not even be necessary to go to a table matching stage. It may be possible to make a decision about a packet’s ultimate destination immediately upon evaluating a certain field within the packet using a simple conditional statement (if-statement, etc).

3.2.12 Egress Processing

Pipeline processing completes when a pipeline stage has finished executing without sending the packet to a new stage. At this point the packet leaves the pipeline and enters egress processing.

First, all actions written the context’s action list are executed. Then the data plane will check the egress port field in the context. If this has been set, then the data plane will queue the packet up to be forwarded through that port. At this
point, processing on the packet has terminated. Otherwise, the packet is dropped and processing terminates on that packet.

3.3 Application Life Cycle and Execution

The life of the Steve application begins when it is loaded by the abstract machine. The machine and application communicate through an ABI described in Chapter 7. The machine calls the application function `ff_load` which statically constructs and initializes flow tables and their respective entries.

To send a packet through pipeline processing, the machine calls the application function `ff_process`, passing it the context constructed during ingress processing. Steve pipeline processing follows a run-to-complete model of execution. Once a packet enters the pipeline, each stage is consecutively applied to the packet. Once no more stages are applied, the packet exits the pipeline, goes through egress processing, and the next packet enters. How a packet transitions between stages is implementation specific. Either the machine controls how the context is passed to the next stage, or the application directly passes the context to the next stage. With the latter, the compiler is able have full, static knowledge of flow control.

The machine may end the application’s life cycle by calling `ff_unload`. This causes all flow tables, flow entries, and any additional resources used by the application to be released.
3.4 Event Handling

Event handlers serve as a function interface into the switch’s control plane. As depicted in Figure 3.1, when a pipeline cannot deal with a packet, it raises an event which is handled by an event handler executed in the control plane. This event handler takes a copy of the context which raised the event as an argument and performs some processing. Event handlers may then choose to make calls back into the data plane’s pipeline to modify its behavior to accommodate for future packets.

An event handler is not guaranteed to execute in the typical run-to-complete model. It is designed to be executed asynchronously by the runtime environment. Certain slower operations which can bottleneck the run-to-completion pipeline are typically performed in event stages. This can involve inserting and removing flow entries to change the pipeline, logging through use of C library functions, etc.

Since the control plane is not bound by the same processor limitations it may execute a wider range of operations supported by general purpose processors (albeit slower) such as those found in a typical C program. However, it is not a guarantee that all switch control planes have the full range of general purpose CPU operations.

3.5 Ports

A switch must have a number of ports to send and receive data. Ports may be *physical* or *logical*. A physical port is a hardware interface on the system. A logical port is a software defined port which may map to multiple physical ports and include
additional abstractions. Additionally, a switch must support a number of reserved ports in order to run a Steve application. These ports must always be present on the system. The required ones are:

- **Drop.** Forwarding to this port causes the packet to be dropped. Its resources are released.

- **All.** Forwarding to this port causes a copy of the packet to be forwarded to every non-reserved port on the system.

- **Flood.** Forwarding to this port causes a copy of the packet to be forwarded to every non-reserved port on the system other than the packet’s ingress port.

- **Controller.** This port acts as a logical communication channel between data plane and control plane. When raising an event, the context is dispatched to this port, which then dispatches the context to the appropriate event handler.

- **Reflow.** Forwarding to this port sends a packet back into ingress processing. All of its context data is released and reconstructed.
CHAPTER IV
STEVE PROGRAM EXAMPLES

This chapter will describe some basic networking applications written in Steve. Four examples are presented: a basic MAC learning bridge, an IPv4 learning switch, a stateless firewall, and a simple wire. These examples are explained from a very high-level perspective, omitting many of the semantic and syntactic details of the Steve language. Instead, this chapter will frequently refer to sections in Chapter 5 which explains Steve syntax, semantics, and use cases in much greater detail.

4.1 A Basic MAC Bridge

The MAC (Ethernet) learning switch, also known as a MAC bridge, forwards packets based on their MAC addresses. Every time the bridge receives a new Ethernet frame, it uses a lookup table to map the source MAC address of the frame to the ingress port that received the frame. When it gets a new frame, it can check the destination MAC address of that frame and know which port it forwards to using the lookup table. If it has not learned that mapping yet, it floods the packet. The learning component prevents the bridge from having to flood all packets like a hub.
The first step in writing any Steve application is defining the layouts. Layouts, which are the topic of Section 5.2, are a way of defining the structure of a header. A bridge only concerns itself with the Ethernet header.

```plaintext
layout ethernet {
    dst : uint (48);
    src : uint (48);
    type : uint (16);
}
```

The next step is writing a decoder, which is the topic of Section 5.3. Since this application is only concerned with the Ethernet header, it will only need an Ethernet decoder.

```plaintext
decoder start eth_d(ethernet) {
    extract ethernet.dst;
    extract ethernet.src;
    goto learn; // Proceed to the first table stage.
}
```

The layout rule for this decoder is `ethernet`. This decoder chooses to extract the source `ethernet.src` and destination `ethernet.dst` MAC addresses. There is no reason to extract `ethernet.type` because this application does not need to know about the Layer 3 protocol. The `start` keyword indicates that this is the first pipeline stage every packet will go through.

After that, the bridge will need flow tables for making forwarding decisions, which are the topic of Section 5.4. Specifically the bridge will need two flow tables: a learning table and a forwarding table. Using two tables will be typical of almost all learning applications. Both tables will start out with nothing but a miss case.
all, before an application runs, it has not learned anything yet. The learning table (learn), will look like the following.

```c
// This table will cause new addresses to be learned.
exact_table learn(ethernet.src) {
    miss -> {
        raise learn_mac; // Raise an event to learn the flows.
        goto forward; // Then send to the forwarding table.
    }
}
```

The learning table is responsible for observing and learning which source MAC addresses map to which port interfaces. It will thus match on `ethernet.src`.

To insert the necessary flow entries, the `learn` table raises an event called `learn_mac`. Events cause a copy of the context to be sent to the control plane to be dealt with by event handlers which are the topic of Section 5.7. The contents of this event handler will be presented a little later in this section. Inserting and removing flow entries will typically be done through event handlers as these actions are slow and bottleneck pipeline processing. After this event is raised, the learning table sends the packet to the forwarding table (forward).

```c
// This table ultimately decides which packet to forward on.
exact_table forward(ethernet.dst) {
    // Flood any packet whose MAC dst hasn’t been learned yet.
    miss -> { output flood; }
}
```

The forwarding table contains flow entries which map MAC addresses to ports. It will lookup destination MAC addresses and ultimately forward the packet. The forwarding table thus matches on `ethernet.dst`. If the table has yet to learn the MAC address, it floods the packet to all ports.
The `learn_mac` event handler is where the actual learning happens, and is thus the most important snippet of code in this application. Event handlers in Steve must explicitly state what fields are extracted before the corresponding event can be raised. This event will require `ethernet.src` – the address being learned.

```plaintext
event learn_mac requires (ethernet.src) {
    // Prevent the learn table from raising this event twice
    // for the same MAC address.
    insert into learn
    { ethernet.src } -> [timeout = 60] { goto forward; }

    // Establish the MAC address to port mapping.
    insert into forward
    { ethernet.src } ->
    [timeout = 60, egress = in_port] {
        // Future packets will be forwarded to
        // the current packet’s ingress port.
        output egress;
    }
}
```

Within the body of the event handler, `ethernet.src` is used like a variable. All extracted fields can be used this way inside decoders, tables/flow entries, and event handlers under certain circumstances described throughout Chapter 5.

This event handler inserts two flow entries. The first flow entry is inserted into the `learn` table. This entry prevents the same MAC address from raising the `learn_mac` event more than once. Instead, the new flow entry sends the packet directly to the `forward` table.

Note the syntax of each flow entry. Match field values appear as a comma-separated list within the block ({ }) before the ->. Steve chooses to represent flow entry actions as a lambda closure following the ->. The action sequence enclosed
by the block ({}{}) following the -> is a lambda function which contains actions to execute on packets which match the flow entry. The named list of properties within [ ] is the environment which the lambda function closes over.

Properties provide predefined, additional, stateful information about a flow entry. The timeout property in this first flow entry indicates that the entry will be removed from the table after 60 seconds. Timeouts are included on these flow entries to ensure that the mapping is not permanent. This allows the device to update its MAC address to port mappings over time if network topology changes.

The second inserted flow entry is where the application learns the new MAC address to port mapping. Recall that the forward table matches on ethernet.dst. Here, the application inserts a flow entry with the current packet’s ethernet.src value into forward. This means that all future packets whose ethernet.dst equals the current packet’s ethernet.src will match this inserted flow entry. The egress property is set equal to the current packet’s ingress port. The flow entry’s body subsequently uses output egress to output matched packets to the current packet’s ingress port. The reasoning behind the egress property can be found in Section 5.4.2.

To summarize, the second inserted flow entry ensures that all future packets whose ethernet.dst field match the current packet’s ethernet.src field will be forwarded to the current packet’s ingress port.
4.2 The IPv4 Learning Switch

The IPv4 learning switch is not too different from the MAC learning switch. Instead of learning and forwarding from MAC addresses, this application will use IPv4 addresses.

To start, the layouts must be defined. The Ethernet layout from the MAC bridge in Section 4.1 can be reused. In addition, the IPv4 layout is also needed. The options field is omitted because Steve currently does not support dynamic-length fields. Also note fields that are not byte-aligned are merged to achieve byte-alignment because Steve currently only supports byte-aligned fields. For example, version and ihl (both 4 bit fields) have been merged into one field. Bit manipulation operations may be used to recover their values.

```
layout ipv4 {
    version_ihl : uint(8);
    dscp_ecn : uint(8);
    len : uint(16);
    id : uint(16);
    fragment : uint(16);
    ttl : uint(8);
    protocol : uint(8);
    checksum : uint(16);
    src : uint(32);
    dst : uint(32);
}
```

An Ethernet decoder is needed again, however, this time the MAC addresses are not needed. Instead, the ethernet.type field is needed to confirm this is an IPv4 packet. If it is, the packet is sent to the IPv4 decoder.

```
decoder start eth_d(ethernet) {
    extract ethernet.type;
    if (ethernet.type >= 0x600)
        match (ethernet.type) {
```
The IPv4 decoder will need `ipv4.src` and `ipv4.dst` in order to learn them, `ipv4.version_ihl` to correctly advance past the IPv4 header, and `ipv4.ttl` to decrement. The rest are extracted to verify the checksum.

```plaintext
def decoder ipv4_d(ipv4) {
    extract ipv4.version_ihl;  // Use this to get header length.
    extract ipv4.dscp_ecn;
    extract ipv4.len;
    extract ipv4.id;
    extract ipv4.fragment;
    extract ipv4.ttl;  // Use time-to-live to decrement
    extract ipv4.protocol;
    extract ipv4.checksum;
    extract ipv4.src;
    extract ipv4.dst;
    // Calculate checksum
    var checksum : uint(16) =
        ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, ipv4.len,
                       ipv4.id, ipv4.fragment, ipv4.ttl,
                       ipv4.protocol, ipv4.src, ipv4.dst);

    if (checksum != ipv4.checksum)
        drop;
    if (ipv4.ttl == 0)
        drop;

    set ipv4.ttl = ipv4.ttl - 1;  // Decrement time-to-live.
    // New checksum with changed time-to-live.
    set ipv4.checksum =
        ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, ipv4.len,
                       ipv4.id, ipv4.fragment, ipv4.ttl,
                       ipv4.protocol, ipv4.src, ipv4.dst);

    // Goto the learn table after advancing view index by IHL.
    goto learn_advance (ipv4.version_ihl & 0x0f) * 4;
}
```
This decoder will drop all packets with bad checksums or whose time-to-live has expired. The function used to calculate the checksum has been elided for brevity. It will decrement $\text{ipv4.ttl}$ and compute a new checksum before sending the packet to table matching. The \texttt{advance} specifier is used to move the context’s view index by the length (in bytes) of the IPv4 header. This is required because IPv4 is a dynamic length header.

Two tables, learning (\texttt{learn}) and forwarding (\texttt{forward}), are needed just like in the MAC bridge example. Their purposes remain largely unchanged, except instead of MAC addresses they will work with IPv4 addresses.

\begin{verbatim}
exact_table learn(ipv4.src) {
  miss -> {
    raise learn_ip;
    goto forward;
  }
}
\end{verbatim}

Here, the \texttt{learn} table matches on $\text{ipv4.src}$. It will raise the event which causes the IPv4 address to be learned. The purpose of this table is to prevent the same IPv4 address from raising the learn event more than once.

The \texttt{forward} table matches on $\text{ipv4.dst}$. This table will establish IPv4 address to port mappings. It will forward all packets whose destination IPv4 address it has learned. Any IPv4 addresses not yet learned are flooded by default.

\begin{verbatim}
exact_table forward(ipv4.dst) {
  miss -> { output flood; }
}
\end{verbatim}
Lastly, the `learn_ip` event must be defined. It is largely the same as the `learn_mac` event from the learning switch.

```plaintext
event learn_ip requires(ipv4.src) {
    // This first entry prevents the same address from causing
    // this event twice.
    insert into learn
    { ipv4.src } -> [timeout = 30] { goto forward; };

    // Establish the IP address to port mapping..
    insert into forward
    { ipv4.src } ->
    [timeout = 30, egress = in_port] {
        output egress;
    };
}
```

The first inserted flow entry prevents the same IPv4 address from being learned more than once. The second flow entry inserts the forwarding rule into the `forward` table. Any packet whose `ipv4.dst` is equal to the current packet’s `ipv4.src` will be forwarded to the current packet’s ingress port.

4.3 A Stateless Firewall Extension

Routers will often have stateless firewalls or packet filters installed. Stateless firewalls maintain a set of rules. These rules only allow packets through if their destination port matches certain port numbers. This firewall example will block all non-HTTP/HTTPS requests on TCP or UDP. This firewall will be an extension to the IPv4 switch from Section 4.2.

The `ethernet` and `ipv4` layouts from the IPv4 switch will remain the same. In addition to these, two transport layer layouts must also be defined: UDP and TCP.
The UDP header contains four fields: source port, destination port, header length, and checksum [35]. The UDP layout is defined as follows.

```plaintext
layout udp {
    src : uint(16);
    dst : uint(16);
    len : uint(16);
    checksum : uint(16);
}
```

The TCP header contains eight fields: source port, destination port, a sequence number, an acknowledgment number, control information (combining data offset and TCP flags), a window size, a checksum, and an urgent pointer number [36]. The dynamic length options field is excluded. The TCP layout is defined as follows.

```plaintext
layout tcp {
    src : uint(16);
    dst : uint(16);
    ack : uint(32);
    seq : uint(32);
    // control : TCP len: 0-3, Reserved: 4-9, Flags: 10-15
    control : uint(16);
    window : uint(16);
    checksum : uint(16);
    urgent_ptr : uint(16);
}
```

The final line of the IPv4 decoder, `ipv4_d`, from Section 4.2 will be replaced so that the packet is now sent to a TCP or UDP decoder rather than to the learning table.

```plaintext
decoder ipv4_d(ipv4) {
    // ipv4 decoder code from before...
    // Changing the final line to dispatch to new decoders.
    var hdr_len : uint(8) = (ipv4.version_ihl & 0xf) * 4;
    match (ipv4.protocol) {
```
A match statement is used on the ipv4.protocol field. If it is equal to 0x06, then the next header to be decoded is TCP. If it is 0x11, then the next header is UDP. A flow table may have also been used to make this decision, however, here it is not strictly necessary.

The TCP and UDP decoders are trivial. All this firewall is concerned with is their destination port. This firewall will ignore performing checksum validation for the sake of simplicity.

```c
decoder udp_d(udp) {
  extract udp.dst;
  goto udp_filter;
}

decoder tcp_d(tcp) {
  extract tcp.dst;
  goto tcp_filter;
}
```

Once the destination port is extracted, each decoder will send the packet to a filtering table. Here, the advance specifier is not attached to the goto tcp_filter action. Normally, it would be, however, if one does not expect another decoder later on, it is acceptable to leave it off.

```c
// Only learn/route if the port# is 80 or 443
exact_table udp_filter(udp.dst) {
  { 80 } -> { goto learn; }
  { 443 } -> { goto learn; }
  miss -> { raise log_udp_port; drop; }
}
```
exact_table tcp_filter(tcp.dst) {
    { 80 } -> { goto learn; }
    { 443 } -> { goto learn; }
    miss -> { raise log_tcp_port; drop; }
}

Each filtering table will only allow packets whose destination ports are 80 (HTTP) and 443 (HTTPS) to be forwarded. All others are logged by an event handler (whose definition can be found in Section 5.7) and dropped. Additional flow entries may be added depending on the purpose of the firewall. From here, the firewall will send to the learning and forwarding tables defined earlier in Section 4.2.

4.4 The Wire

A wire is a network function which has two ports. It receives from one port and outputs out of the other port. The only caveat is that the application is not aware of the ports comprising the wire at first. It must learn that those ports exist.

This example demonstrates a number of more unintuitive features related to ports. First, two uninitialized port variables named p1 and p2 are declared. Uninitialized port variables always compare equal to 0.

Port p1;
Port p2;

The Ethernet layout from prior examples can be used to write a starting decoder. The special thing about this decoder is that it does not care about nor extract any of the fields. It only needs the ingress port. The decoder would look like the following:
decoder start eth_d(ethernet) {
  // Whenever a packet is handled, check if p1 and p2 are set.
  // If neither are, set p1 equal to the ingress port.
  if (p1 == 0 && p2 == 0)
    p1 = in_port;
  // If p1 is set, and p2 isn’t, set p2 to the ingress port.
  else if (p1 != 0 && p2 == 0)
    p2 = in_port;
  // Decide where to forward.
  // If the ingress port is p1, and p2 is set, forward to p2.
  if (in_port == p1 && p2 != 0)
    output p2;
  // If the ingress port is p2, and p1 is set, forward to p1.
  if (in_port == p2 && p1 != 0)
    output p1;
  // If both are not set yet, do nothing and implicitly drop.
}

Here, the two port variables are used to remember what ports exist. The decoder looks at the ingress port and saves it into one of the two variables. Once two ports are “remembered,” forwarding decisions can be made. If a packet comes from p1, then it is forwarded through the only other port, p2, and vice-versa. Before both ports are saved, since no other actions are applied, the packet finishes pipeline processing and gets dropped.

For a wire, it is always advised that one primer packet is sent through both interfaces before any packets are handled, to ensure the application is immediately aware of both ports.
CHAPTER V

THE STEVE LANGUAGE

The Steve Programming Language provides features for defining and connecting the pipeline processing stages and event handlers. Together, these components comprise a network application.

This chapter will more carefully break down syntax from examples in Chapter 4. It will also describe all of the language features for Steve in greater detail with additional use cases. Specifically, this chapter will discuss how headers are represented (Section 5.2), how to write decoders (Section 5.3), how to write flow tables (Section 5.4), how to apply actions to packets (Section 5.5), and how to write event handlers (Section 5.7).

Semantics and limitations of Steve will be mentioned throughout this chapter, but not in complete detail. For the semantic description of Steve, including grammar, typing rules, and other restrictions, see the Reference Guide in Appendix A.

5.1 General Purpose Language Features

Steve supports language features that can be considered general purpose. They are common to most programming languages and are not explicitly for packet processing, though they may prove useful.
5.1.1 Variables

Steve allows for the allocation of both local and global variables. A variable named \( x \) which holds an integer 10 is written as follows.

```plaintext
var x : int = 10;
x = 9;  // Assign a new value.
```

5.1.2 Conditional and Loop Statements

Steve supports two conditional statements for decision making: the `if` statement and the `match` statement. The `if` statement is the same as C.

A `match` statement allows for a decision to be made given a number of possible case values. This is similar to a C-like switch statement, with the only major difference being that there is no *fall-through* behavior. In other words, after the execution of a `case` statement, control jumps out of the `match` statement rather than moving to the next case (i.e. an implied `break`). The condition and labels must be integers just like in C. A `match` statement can be written as follows.

```plaintext
match (x) {
  case 0: x = x + 1;
  // Multiple statements following the label must be
  // enclosed in a block.
  case 1: {
    x = x + 2;
    y = y * x;
  }
  // The default case statement.
  miss: x = 0;
}
```
Steve also supports the \texttt{while} loop which is the same as C-like languages. It also supports the \texttt{break} and \texttt{continue} statements for limited branching abilities inside a loop.

5.1.3 Functions

Steve supports writing simple functions, though the syntax is a little different from C-like languages. A function named \texttt{sum} which takes two integers, \texttt{a} and \texttt{b}, and returned their sum is written as follows. The return type follows the `->` in Steve functions.

\begin{verbatim}
def sum(a : int, b : int) -> int {
    return a + b;
}
def sum(1, 2);
\end{verbatim}

5.1.4 Foreign Functions

By default, all Steve compiled applications are linked against the C runtime library. Steve applications may also be linked against other libraries. Steve programmers may call functions in linked libraries by first declaring them in the Steve application using a \textit{foreign function} with the same function signature.

\begin{verbatim}
foreign def puts(input : char[]) -> int;
puts("Hello, World.");
\end{verbatim}

Here, this foreign function links to the C standard \texttt{puts} function for text output. It may be called like any other function.
5.2 Layouts

A layout is used to describe the structure of a packet header. More specifically, they describe what fields are present, their lengths, the order in which they appear, and their relative offset from the beginning of the header. Decoders use layout information to reason about extracting fields.

A layout declaration (see A.8.1) is used to write a layout. The simplest example to begin with is the Ethernet frame header [37]. The corresponding layout would look like the following.

```plaintext
layout ethernet {
    dst : uint(48); // 48 bits.
    src : uint(48); // 48 bits.
    type : uint(16); // 16 bits.
}
```

Every layout has a name (ethernet) and a sequence of field declarations which describe the fields contained within the header. This layout has three such field declarations (dst, src, and type). Each field declaration has a name and a type that describes valid operations and value ranges for that field. To reduce errors related to working with byte buffers, Steve allows header fields to be represented as fixed-precision, signed or unsigned, integer types. The precision denotes the length of the field in bits. Here, dst and src fields are 48 bits long and type is 16 bits long.

The relative offset of each field is the number of bits it is away from the beginning of the header. The first field will always have a relative offset of 0 bits. The relative offset of each subsequent field is equal to the sum of the lengths of all
fields preceding it. Here, dst has a relative offset of 0 bits, src has one of 48 bits (6 bytes), and type has one of 96 bits (12 bytes).

The field declarations must appear in the order with which they would normally appear in an Ethernet header. Field ordering should always be preserved when declaring layouts. If the ordering is incorrect, decoders will assume a sequence of bits is a certain field when it truly is not.

Not all header structures are as simple as the Ethernet header. Sometimes one must deal with structures inside a header. Steve supports composition of layouts to describe such header structures. The following presents an OpenFlow protocol message which contains such a case [1].

```c
layout ofp_instr_stat_trigger {
    type : uint(16);
    len : uint(16);
    flags : uint(32);
    threshold : ofp_stats;
}
layout ofp_stats {
    reserved : uint(16);
    length : uint(16);
    oxs_field : uint(8);
    padding : uint(32);
}
```

In this example, two layouts are presented: ofp_instr_stat_trigger and ofp_stats. To express the ofp_stats structure within the OpenFlow message, a field named threshold is added to ofp_instr_stat_trigger and its type is given as the name of another layout (ofp_stats). The length of the threshold field would be the sum of the lengths its fields, in this case, 72 bits (9 bytes).
With the way layouts are described and written, it is easy to draw the comparison between layouts and class types (or record types). *Layouts are not classes*. Layouts are much stricter.

First, the *types of fields are restricted*. Fields may only have two types: integer (see A.6.3) and layout (see A.6.8). There may be varying kinds of integer (e.g. precisions, signed, unsigned, etc.), but the precision of each integer must also be *byte-aligned*, that is, a multiple of 8.

Second, the most important distinction is that objects of layout type can never be created. Layouts may not appear as the type of parameters, nor may they appear as return types. All of the following are considered ill-formed.

```plaintext
layout L1 { f1 : uint; f2 : uint(16); }
var x : L1 = 0;  // Error.
def foo(y : L1) -> L1 { ... }  // Error.
```

Layouts are distinct from the headers that they describe. The sole purpose of layouts is to allow decoders to reason about the logical structure of a header in memory. The memory for packets and their headers exist independent of a running Steve application. To create an object of layout type would imply the need to create new headers, which is not currently supported by Steve.

Another important thing to note is that Steve does not currently support dynamically sized types (DST). A DST is a type whose size is predicated upon some value known only during runtime. These DST’s are used to represent fields whose lengths are dynamic. Some examples of dynamic length fields are the `options` fields in IPv4, IPv6 extended, and TCP headers [38, 39, 36].
DST’s are a language feature that will eventually be added, but are outside the current implementation. Because of this, fields whose lengths are dynamic cannot currently be declared, extracted, nor used. The existence and eventual support of DST’s is one of the reasons why objects of layout type cannot be created. This is further discussed in Section 6.2.

The following example presents a case where some of these limitations become relevant – the IPv4 layout.

```plaintext
layout ipv4 {
  version_ihl : uint(8); // Non-byte aligned fields are merged
  dscp_ecn : uint(8); // This is merged, too.
  len : uint(16);
  id : uint(16);
  fragment : uint(16); // Fragment flags and offset merged.
  ttl : uint(8);
  protocol : uint(8);
  checksum : uint(16);
  src : uint(32);
  dst : uint(32);
  // Options field omitted because it is a DST.
}
```

In this example, `version` (a 4 bit field) has to be merged with `ihl` (internet header length) (also a 4 bit field) to achieve byte alignment. The same is true for `dscp` and `ecn`. The `fragment` field, typically composed of three 1 bit flags and a 13 bit fragment offset field, is merged into a single 16 bit field. Bitwise-AND and bit shifting is needed to recover the needed bits. Examples will come up later when an IPv4 decoder is discussed in Section 5.3.3.
5.3 Decoders

Decoders are special purpose functions used to extract fields from a single header. They implicitly take context data structures as arguments and operate on them. A decoder conforms to a layout specification. Layout information allows the decoder to determine where fields are located in a header and how long they are.

5.3.1 The Basic Decoder Form

A decoder is written using a *decoder declaration* (see A.8.2). The most common decoder written is likely the Ethernet decoder. The following is the basic form of an Ethernet decoder.

```latex
decoder start eth_d(ethernet) { ... }
```

A decoder declaration has four important parts: 1) a name (*eth_d*), 2) a *layout rule* (*ethernet*), 3) a body, and 4) the optional *start* keyword.

The decoding process conforms to the layout rule and uses it to reason about the location and lengths of fields within the header it is decoding. Decoder operations are placed in the body delimited by {}. The *start* keyword identifies this decoder as the **starting decoder**.

A starting decoder is the root of the pipeline. By extension there can only be one starting decoder. Every packet must be processed by the starting decoder first. Since Ethernet is the most common Layer 2 framing protocol, it will likely be the root of almost all pipelines.
5.3.2 Extractions

An extract declaration (see A.8.3) in the decoder’s body instructs it to extract the given field. Extracted fields, or extractions, for short, may then be used by the program as a variable. The following example expands the body of the eth_d decoder with extract declarations.

```plaintext
decoder start eth_d(ethernet) {
    extract ethernet.dst;
    extract ethernet.type;
    // ...
}
```

Each extract declaration gives a field name (see A.2) which tells the decoder which field is being extracted. In this example, there are two such extract declarations which instruct the decoder to extract the fields `ethernet.dst` and `ethernet.type`. The compiler generates \((\text{offset, length})\) pairs denoting the relative offset and length of the field within the decoder’s view using the given field names and layout rule.

Only field names which refer to fields in the layout rule may be used. It makes no sense to extract a field not in the header. For example, it would not be possible to extract `ipv4.protocol` in the `eth_d` decoder.

5.3.3 Accessing Extracted Fields

Once a field has been extracted, getting its value is similar to using it as variable. To get the value of an extraction the field access expression (see A.10.3) is used. A field access expression is a field name being used where an expression is expected.
This is similar to using a variable name in an expression to represent the value of the variable.

A typical operation on an Ethernet header is determining which protocol it encapsulates, i.e. the header which comes next. This is indicated by the value stored in `ethernet.type`. The following example demonstrates how the value of the `ethernet.type` extraction might be used.

```c
decoder start eth_d(ethernet) {
    extract ethernet.dst;
    extract ethernet.type;
    if (ethernet.type >= 0x600) {
        // The type determines what header comes next...
    }
    else if (ethernet.type <= 0x05dc) {
        // The type is the length of the entire packet...
    }
}
```

The IEEE Ethernet standard says that `ethernet.type` fields greater than or equal to `0x600` indicate the next header's protocol [37]. Any `type` fields less than `0x05dc` indicate the Ethernet frame's length. Here, field access is used to compare `ethernet.type` to hexadecimal literals in an `if-else` statement to determine the meaning of that field.

Extraction values can also be used in arithmetic operations, comparison operations, bitwise operations, function calls, and can be stored and assigned to variables. The following example presents a trivial IPv4 decoder demonstrating some of these basic operations.

```c
decoder ipv4_d(ipv4) {
    extract ipv4.version_ihl; // Use this to get header length.
```
This example presents a solution for recovering non-byte aligned fields. A bitwise-AND (see A.10.20) is used on `ipv4.version_ihl` with `0x0f` to recover the `ihl` field. The `ipv4.version_ihl` is left-shift by 4 bits to get the `version` field.

Extractions can be passed to functions as well. A convenient use case is calculating the checksum for the IPv4 header. The following example extends the IPv4 decoder with a few more operations.

```plaintext
decoder ipv4_d(ipv4) {
  // ...
  // Calculate a checksum by calling a function.
  var checksum : uint(16) =
    ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, ipv4.len, ipv4.id, ipv4.fragment, ipv4.ttl, ipv4.protocol, ipv4.src, ipv4.dst);
  // Check the checksum against the header’s checksum.
  if (checksum != ipv4.checksum)
    drop;
  // Drop time-to-live expired packets.
  if (ipv4.ttl == 0)
    drop;
  set ipv4.ttl = ipv4.ttl - 1; // Decrement time-to-live
  // The ttl has changed. A new checksum must be calculated.
  set ipv4.checksum =
    ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, ipv4.len,
```
Fields from the IPv4 header are passed to a function named `ipv4_checksum` (whose definition has been elided for brevity). The resulting checksum is compared against the current checksum using an if statement. If they do not compare equal, then the packet is dropped using the `drop` action.

Decrementing the time-to-live (`ipv4.ttl`) is also a common operation. First the time-to-live is checked to see if it is 0. If it is, then the packet’s lifetime has expired and it will be dropped. If the packet is still valid, the field is decremented using simple subtraction and the `set` action. Because a field has been changed in the IPv4 header, the checksum must be recalculated and set with the new checksum.

Field access expressions do have a number of limitations. The following example demonstrates some of them.

```
decoder start eth_d(ethernet) {
    // Error: Cannot use eth.type before its extracted.
    if (ethernet.type >= 0x600) { }
    extract ethernet.type;
    // Error: Cannot assign to a field this way.
    ethernet.type = 0x800;
    // OK: A set action must be used instead.
    set ethernet.type = 0x800;
}

decoder ipv4_decode(ipv4) {
    // Error: This decoder does not decode ethernet.
    extract ethernet.type;
    // Error: ethernet.type was not extracted by this decoder.
    if (ethernet.type == 0x800) { }
}
```
A field access expression can only be used after an extract declaration is made for that field since it is impossible to recover the value of a field which has not been extracted. By extension, they cannot be used inside a decoder that has not extracted that field even if a prior decoder extracted that field. A decoder focuses on exactly one header and has no knowledge of previous headers or extractions. Field access expressions are not the same as variables and may not be assigned to like one. To modify the value of a field, a set action must be used instead.

5.3.4 Rebinding Extracted Fields

The context binding environment allows fields to be rebound with different names. This essentially allows the a field to be referred to using a different name. When extracting multiple fields of the same name, this allows the programmer to disambiguate them.

More commonly it is used to rename fields so that they may be reinterpreted as other fields. For example, assume that an Ethernet frame has a VLAN tag. The ethertype field of the VLAN tag is what actually informs the switch what Layer 3 protocol is encapsulated by Ethernet. A rebind declaration (A.8.4) is used to alias a field with a new name. So for example:

```plaintext
layout vlan {
    tci : uint(16);
    ethertype : uint(16);
}

decoder start eth_d(ethernet) {
    extract ethernet.type;
    match (ethernet.type) {
        case 0x8100: decode vlan_d;
    }
}
```
In vlan_d, the `extract vlan.ethertype as ethernet.type` allows the extraction of `vlan.ethertype` to be referred to as `ethernet.type` in future stages. The `vlan.ethertype` is the original field and `ethernet.type` is the alias field. Consider the following table:

```
extact_table ethtable(ethernet.type) {
    { 0x8100 } -> { ... }
    { 0x86dd } -> { ... }
}
```

Assume that `vlan.ethertype` has a value of 0x86dd. When `vlan_d` sends the packet to `ethtable`, it will match the flow entry whose match field is 0x86dd. Because of the rebind `ethernet.type` refers to the extracted `vlan.ethertype`.

5.3.5 Stage Transition

Once the current stage has finished its work, the programmer may decide to send the packet to another pipeline stage. Section 3.2.11 describes that decoding and table matching stages can be chained together in a number of flexible ways. Any stage may move a packet to a new decoder or a new table.

To move to another decoding stage, the `decode` action (see 5.5.1) is used. The following example bridges the Ethernet and IPv4 decoders declared in earlier examples. Extract declarations are elided for brevity.
The **match** statement is used to check if `ethernet.type` is equal to 0x800. If it is, then it confirms the next header is IPv4 and the **decode** action moves the packet to the IPv4 decoder.

To transition to a table matching stage, a **goto** action (see 5.5.2) (not to be confused with a C-like **goto**) is used.

```plaintext
decoder ipv4_d(ipv4) {
    // ...
    var ihl : uint(8) = (ipv4.version_ihl & 0x0f) * 4;
    // Advance specifier shifts the view by N bytes.
    goto tcp_filter advance ihl; // tcp_filter names a table.
}
```

In this example, the **goto** action sends the packet to a hypothetical table named `tcp_filter`. Details about writing tables follows in Section 5.4. The most important thing to notice from this example is the **advance** specifier.

Section 3.2.5 explains that a decoder shifts the **view** of a packet before moving to the next stage. That shift is by the length of the header. IPv4 headers are dynamic in length. Even though Steve does not currently support extracting dynamic length fields, one must still account for them. To correctly do this, the **advance** specifier is applied to explicitly shift the view by the given number of **bytes**. Here, the `ihl`
calculation determines the number of bytes in the header. The assumption is made that all headers are byte-aligned, therefore advancing by a number of bytes (rather than bits) is appropriate.

The *advance* specifier may appear on both *goto* and *decode* actions. However, it may only appear in a decoder, as decoders are the only stage concerned with views.

A stage is *complete* once it executes a *decode* or *goto* action, or finishes executing without either action being executed. These two actions are similar in semantics to a function return in that the stage will not execute anything after them. If no stage transition happens at all, the packet exits pipeline processing and enters egress processing.

Actions appended to the context’s action list are executed before a forwarding decision is made. The original packet is forwarded based on the egress port field stored within the packet context. This field is set when executing an output action that has been written to the context’s action list (see 5.5.8) for an example). If this field was not set, then the packet is implicitly dropped.

### 5.4 Tables

The next stage is the table matching stage which uses the flow table mechanism described in Section 3.2.8 to direct network flows. Flow tables are decision tables that are built into the language.
5.4.1 The Basic Table

The following example presents the basic form of a table which groups TCP packets by destination port. The definition of the TCP layout has been elided here, but may be found in Section 4.3.

```
exact_table tcp_filter(tcp.dst) {
    // Flow entries...
}
```

Each flow table is comprised of three parts: 1) a name (`tcp_filter`), 2) a key, and 3) a set of flow entries. Additionally, there may be three kinds of flow tables: exact, prefix, and wildcard. This is an exact match table (Steve currently only supports exact match tables).

A table’s key provides a comma-separated list of field names, known as key fields, which indicate which fields a table matches on. They are the equivalent of decision attributes. The `tcp_filter` table matches on a single field (`tcp.dst`).

It is important to note that a goto action may only dispatch a packet to a table if the key fields have been extracted in prior decoders. This ensures that the key fields actually exist before matching begins. If they have not been extracted, the program is ill-formed. How the compiler checks this is explained in Section 6.1.3.

Flow entries define the criteria for a network flow, and what actions are taken on packets belonging to those flows. They are the rules of a decision table. To write a flow entry, a flow entry declaration (see A.8.5) is used. The following example presents three flow entries: two regular ones followed by a special one known as the miss case.
A flow entry declaration has two parts: 1) match fields and 2) an action sequence (also called the body of the entry). Match fields appear as a comma-separated list of expressions in the brace-enclosed block ({} ) before the ->. The first two flow entries have a single match field each – 80 and 443, respectively.

Match fields are values which correspond to the table’s key fields. When a packet is matched against a table, the table compares the packet’s fields with the match fields of each flow entry. A packet matches a flow entry if each field (which is part of the table’s key) in the packet matches each corresponding match field in the flow entry.

The action sequence is given in the brace-enclosed block ({} ) after the ->. When a packet matches a flow entry, actions in this sequence are executed. In this example, if a packet’s TCP destination port field is equal to 80, then it will match the first flow entry and it will be sent to another flow table named forward. The same is true for packets whose TCP destination port field is equal to 443.

The third flow entry is a special entry known as the miss case. The miss case uses the keyword miss rather than providing match fields. If no other flow entry in a table can match a packet, the miss case is used. A table may only have one miss case. If one is not given, an implicit one exists which is the equivalent to the one in the example. That is, by default miss cases drop a packet.
Flow entries declared within a flow table body are known as *initial flow entries*. They get installed when a Steve application is loaded by the runtime. Flow entries may also be added to a flow table after it has begun processing packets. An example of adding and removing flow entries can be found in Section 5.5.5 and 5.5.6, respectively.

### 5.4.2 A More Complex Table

The `tcp_filter` table from Section 5.4.1 is only the most basic of flow tables. Flow tables can have more complicated use cases. Not all flow tables will match on a single field. In fact, most flow tables will match on many fields. The following table declaration demonstrates this.

```plaintext
exact_table ip_proto (ipv4.fragment, ipv4.protocol) {
    { 0x0, 0x11 } -> { decode udp_d; }
    // And so on...
}
```

This table classifies packets into flows based on their fragment value and the IP encapsulated transport layer protocol. This flow entry groups non-fragmented, UDP packets.

Flow entries may also have *properties*. Properties are additional information stored alongside flow entries. Steve currently supports two properties: timeout and egress. The following example extends the previous `ip_proto` table with a new flow entry using the timeout property.

```plaintext
exact_table ip_proto (ipv4.fragment, ipv4.protocol) {
    { 0x0, 0x06 } -> [timeout = 1000] { decode tcp_d; }
    // And so on...
}
```

72
Properties are given in a comma-separated list within the block ([ ] ) imme-
diately following the ->. If the timeout property is set, the flow entry will be ejected
from its table after a given number of seconds. This value may be between 1 and
65,535.

The egress property is used to store a reference to a port. The reason this
is needed is not immediately obvious without context. Suppose a flow entry is being
inserted into a table using an insert action.

```
insert into tcp_filter
{ 22 } -> { output in_port; }
```

Within the inserted flow entry, there is the output in_port action. The
problem is that the meaning of in_port is ambiguous. Does it refer to the current
packet’s ingress port, or the ingress port of future packets that will match this flow
eentry? Because the context is an implicit object within all of these stages, this
ambiguity can arise.

To solve this problem, flow entries may capture the current packet’s ingress
port (or any port name in scope) with a new name (egress). So for example:

```
insert into tcp_filter
{ 22 } -> [egress = in_port] { output egress; }
```

The output egress action now unambiguously says to forward all future
matched packets to the current packet’s ingress port. This is particularly useful for
learning applications such as the MAC learning switch in Section 4.1).
5.4.3 Flow Entries as Lambda Closures

The reason flow entry properties work is because the properties block and body are a lambda closure. The body defines the lambda function which executes when a packet matches; the properties block is the environment being closed over.

5.4.4 Table Requirements

Key fields are typically used to match specific values or sets of values. However, some flow tables may need fields extracted, yet not necessarily care what the values of those fields are. For example, an IPv6 table may want to decrement the hop-limit field (equivalent to time-to-live). For these cases, there is the requires specifier. The following example is a table which matches on one field and also requires ipv6.next_hop.

```
exact_table ipv6_proto(ipv6.next_header)
  requires (ipv6.hop_limit)
  {{ 0x01 } -> {
    set ipv6.hop_limit = ipv6.hop_limit - 1;
    decode icmp_d;
  }}
```

The requires specifier is a comma-separated sequence of field names which must be extracted before the table may match the packet. Each required field can be thought of as a wildcard value (*).

Tables implicitly require their key fields so placing key fields in the requires specifier is redundant and has no effect. Requirements are used by the language to reason about what fields must be extracted before it is safe to reach a given stage.
It forces the programmer to be aware that they have extracted a field before using it, otherwise the program is ill-formed. How this is enforced is explained in Section 6.1.3.

5.4.5 When To Use a Flow Table

There are certain scenarios that must be considered before using a flow table. Flow tables have certain distinct advantages over static decision structures (if or match).

*Tables can match on one or more fields at once.* The more fields required in the decision making process, the more complex using nested decision structures gets. Tables can also match on a packet’s ingress port (in_port) and physical ingress port (in_phys_port) fields (described in Section 5.5.3).

*Tables can match on and use fields from different headers.* Unlike decoders, tables have access to all extractions. The only limitation is that field access in the table’s flow entries only works on key fields or required fields. For example, the following is a valid table.

```
exact_table T(in_port, in_phys_port, ethernet.dst, ipv4.dst)
{ ... }
```

*Flow entries can be added and removed from tables using the appropriate actions.* This allows decision making on packets to change dynamically during runtime. It is obviously impossible to add new branches to conditional statements. The ability to add, or learn, new entries allows us to write applications which can evolve, such as learning switches and routers.
In some cases conditional statements are preferred over tables. Table matching can be slow. If none of the advantages of using a table are needed, it is almost always more beneficial to use a conditional statement. Table matching is inherently slower than conditional branching and should actually be avoided when possible.

5.5 Actions

Actions modify packets, action lists, and pipeline state. Steve supports ten actions with more anticipated in the future. Actions may appear in decoders, flow entries, and event handlers.

5.5.1 Decode Action

The decode action is used to move a packet from the current stage to a decoding stage. This action was present throughout a number of examples so far. For example:

```plaintext
decode ipv4_d; // ipv4_d names a decoder
```

There is also an optional advance specifier which is used if the view of the packet must be explicitly shifted by some special number of bytes. For example:

```plaintext
decode udp_d advance (ipv4.version_ihl & 0x0f) * 4;
```

The advance specifier may only be attached if the action is executed by a decoder. Only decoders are responsible for view shifts.

5.5.2 Goto Action

The goto action is used to move a packet from the current stage to a table matching stage. For example:
goto t1;  // t1 names a table

Similar to the decode action, the goto action also supports an optional advance specifier. For example, if the current decoder is for IPv4, and the table is named t1, the action would be written as:

goto t1 advance (ipv4.version_ihl & 0x0f) * 4;

The advance specifier may only be attached if the action is executed by a decoder. Only decoders are responsible for view shifts.

5.5.3 Output Action

Output actions forward a copy of the current packet to a port. For example, the following forwards to the reserved flood port.

output flood;

More on ports and how to use the output action with them can be found in Section 5.6. The flood port is only one of a number of reserved ports which Steve supports.

There are certain implications to the output action forwarding a copy of the packet. Output actions do not end pipeline processing on a packet. Multiple output actions can be executed in the same processing stage. Pipeline processing will continue on the original packet after the output action.

The original packet is forwarded after pipeline processing completes, during egress processing. At that point the context’s action list is executed. Output actions
which have been written to the action list modify the egress port field in the context when executed. This field determines where the original packet gets forwarded.

5.5.4 Drop Action

A packet can be dropped by the Steve application using the drop action. The drop action terminates the pipeline processing of a packet.

\[ \text{drop;} \]

5.5.5 Insert Flow Action

This action is for inserting flow entries into a table. This allows forwarding decisions to evolve at runtime. The syntax is intentionally similar to SQL so that users are immediately familiar with the concept.

Flow entries can be inserted with constant key values and no properties. Here, a flow entry is inserted into the table presented in Section 5.4.2.

\[
\text{insert into ip.proto}
\{ 0x0, 0x89 \} -> \{ \text{decode mpls_d;} \};
\]

They can also be inserted with recent extraction values and with optional properties.

\[
\text{insert into ip.proto}
\{ \text{ipv4.fragment, ipv4.protocol} \} ->
[\text{timeout = 1000}] \{ ... \};
\]

If a new flow entry’s match fields already exist in the table, the old flow entry is replaced by the new flow entry. A miss case may be inserted into a table as well.

\[
\text{insert into ip.proto}
\text{miss} -> \{ \text{output flood;} \};
\]
5.5.6 Remove Flow Action

A flow entry can be removed from a table by providing match field values and the name of the table to remove the flow entry from. This can be done with constant values or extraction values of the current packet. If the flow entry with those match field values does not exist, nothing is done.

```python
remove from ip_proto { 0x0, 0x89 };
remove from ip_proto {ipv4.fragment, ipv4.protocol};
```

Miss cases can also be removed from tables. When a miss case is removed, it is replaced by the default miss case (which drops the packet).

```python
remove from ip_proto miss;
```

5.5.7 Set Action

A `set` action can be used to write to any extracted field within a packet. The `set` action is guaranteed to never overflow or underflow when writing bytes into the packet buffer. The correct amount of bytes is always written, and always into the correct position in the packet. To enforce this, the value being written must have the same integer type (specifically precision) as the field being written to. If they mismatch, implicit integer conversion (see A.7.2) ensures that the value being written gets converted to the correct type.

The time-to-live field from an IPv4 header may be set as follows.

```python
set ipv4.ttl = ipv4.ttl - 1;
```
The set action is only valid if the field access expression is valid in that stage. For decoders, this means the field has to have been extracted first. For tables, this means the field must be a key field or a required field. For events, the field must be a required field. This ensures that a field has been extracted before it can be written to. It's obviously not possible to write to a field if its location and length are unknown.

5.5.8 Write Action

The context data structure described in Section 3.2.2 keeps an action list. Actions get written to the action list using the write action. Written actions get executed once pipeline processing completes and the packet enters egress processing (described in Section 3.2.12). Only two actions may be written to a packet right now: output and set.

```
write set ipv4.ttl = ipv4.ttl - 1;
write output reflow;
```

The written output action has a slightly different semantic from the immediately applied output action. When immediately applied, the output action forwards a copy of the packet. The written output action changes the egress port field in the packet context when executed. This field ultimately decides where to forward the original packet.

5.5.9 Clear Action

The clear action removes all actions from the context’s action list.

```
clear;
```
5.5.10 Raise Action

A `raise` action is used to raise an `event`. Events are used to signal the control plane that an exceptional situation has happened. *Event handlers* (described in Section 5.7) are used to deal with these events.

A `raise` action sends a copy of the context to a reserved port called the controller port that connects to the control plane. The controller port then offloads the context to the appropriate event handler. For example:

```plaintext
raise log_tcp_port;
```

Here, `log_tcp_event` names the event handler which will be executed. Its definition can be found in Section 5.7. The execution is asynchronous to the pipeline, meaning the pipeline will not wait for it to complete before executing the next action.

Because an event handler may choose to re-enter the context into the pipeline, the `raise` action may also use an `advance` specifier when done in a decoder. However, decoders should rarely, if ever, explicitly raise events.

5.6 Ports

Steve supports limited access to ports. There are two categories of ports to consider when working with Steve: *reserved ports* and *non-reserved ports*.

5.6.1 Reserved Ports

Reserved ports are named ports which are always present on the system. Some ports may be directly forwarded to using the `output` action. Other reserved ports
are forwarded to implicitly by other actions. The following reserved ports may be forwarded to by the output action.

- **All port.** Forwarding to the all port will forward copies of the packet to every port on the system.

- **Reflow port.** Forwarding to the reflow port will send the packet back into ingress processing. From there it will be processed again by the pipeline from the beginning.

- **Flood port.** Forwarding to the flood port sends copies of the packet to all ports on the system except the packet’s ingress port.

Each of these reserved ports can be accessed using a reserved keyword. In the following, the output action is used to send a packet to these three ports.

```plaintext
output all;
output reflow;
output flood;
```

The following reserved ports are forwarded to implicitly by other actions.

- Packets are forwarded to the drop port by the drop action. Packets which are dropped get deleted.

- Packets are forwarded to the controller port when the raise action is used. The controller port is logically connected to the controller (or control plane). This controller is responsible for dispatching and executing the appropriate event handlers.
5.6.2 Non-reserved Ports

Non-reserved ports are not guaranteed to be present on all systems. These ports can be further categorized into *physical* and *logical* ports. A physical port is a hardware interface on the system. A logical port is a software defined port which may map to multiple physical ports and include additional abstractions.

Steve applications currently do not support a way of directly discovering all these ports and their capabilities. Steve applications can indirectly learn about these ports by observing the ingress ports of packets passing through the pipeline. There are two reserved keywords for getting the logical ingress port and physical ingress port of a packet.

```plaintext
in_port; // The logical ingress port.
in_phys_port; // The physical ingress port.
```

5.6.3 Port Variables

Port variables can be used to *remember* ports for later usage. They are written using *port declarations* (see A.8.7). They work as port variables.

```plaintext
Port p1;
Port p2;
```

Other ports can be assigned to them. This does not copy the port. It just saves a handle to that port inside the port variable.

```plaintext
p1 = in_port;
p2 = in_phys_port;

output p1; // Forward to these ports.
output p2;
```
Here, the ingress ports of a packet are saved to port variables \texttt{p1} and \texttt{p2}. An \texttt{output} action may specify a port variable, in which case the packet is forwarded to the port whose handle is stored by the port variable.

5.7 Events

An \textit{event handler} is a processing stage executed in the control plane outside the regular run-to-completion pipeline. They are functions that operate on packet contexts. Event handlers provide an interface for the pipeline to access the control plane. Certain slower operations are best performed by event handlers. Since they execute asynchronous to the data plane’s pipeline, they do not bottleneck it. Specifically, inserting and removing flow entries are best executed inside events. These operations are multiple times slower than any other actions in Steve.

Another advantage of event handlers is that they are not subject to the same limitations on operations as flow tables. They may call functions, write to files through the C standard library, allocate variables, etc., as well as execute actions.

To write an event handler, an \textit{event declaration} (see A.8.6) is used. The following is a simple example.

```c
event log_tcp_port requires (ipv4.dst, tcp.dst) {
    var file : uint(8)& = fopen("blocked.txt", "a");
    fprintf(file, "Blocked packet from \%x to TCP port \%d",
            ipv4.dst, ipv4.dst);
    fclose(file);
}
```
This event handler is used to log IP destination addresses and TCP destination ports. It is useful for firewall applications. Event declarations have a \texttt{requires} specifier like tables. An event may only be raised if all fields listed in its \texttt{requires} specifier have been extracted.

The \texttt{raise} action is used to send a copy of a context to an event. Any changes made in the event handler do not modify the original packet or its context.

Steve provides the advantage of being able to write event handlers in the same language as the pipeline. Steve event handlers are thus subject to the same safety guarantees applied to pipeline stages. This is possible because Steve targets switches that do not require centralized control planes.
CHAPTER VI

STEVE SAFETY AND LIMITATIONS

This chapter discusses the safety guarantees enforced in individual stages and the pipeline as whole. It will also expand upon the limitations briefly mentioned in the tutorial from Chapter 5.

6.1 Pipeline Guarantees and Constraints

Steve makes guarantees about the logical correctness and safety of pipeline composition using a constraints system. These guarantees are enforced by ensuring each Steve pipeline has two properties: progress and memory safety. Any pipeline which does not have these properties is ill-formed.

6.1.1 Pipeline to Graph Conversion

To check these properties, a directed graph, known as a transition graph, is constructed over stage transitions within the pipeline. Each property thus becomes a graph evaluation algorithm on this graph.

Let the transition graph $G = (V, E)$ be defined as follows. $V$ is the set of vertices representing declarations of stages in the program: decoders, flow tables, and event handlers. $E$ is the set of edges derived from decode, goto, and raise actions.
found in those definitions. There exists a directed edge \((s, d)\), where \(s\) is the stage containing the action and \(d\) is the destination stage.

The Steve language defines exactly one entry point to the packet processing pipeline – the starting decoder. Any vertex not reachable from the starting decoder is excluded from analysis since it will never be executed.

6.1.2 Progress

The *progress* property guarantees that packet processing always terminates. Though the syntax allows the programmer to construct pipelines that would not terminate, construction is restricted semantically. Such guarantees are excluded for while loops and function calls due to the halting problem.

A Steve program terminates if and only if the transition graph is a directed acyclic graph (DAG), otherwise it contains cycles between stages which potentially result in non-termination. To check this, simply generate a topological ordering of vertices in the stage transition graph. If no such order can be constructed, the pipeline does not guarantee progress (i.e., the graph admits a cycle), and the program is ill-formed.

6.1.3 Memory Safety

The *memory safety* property ensures that memory locations that are not declared or allocated by the program are not accessed. For packets, this means that packet memory must be checked to guarantee existence and valid construction. This work
focuses on existence; previous Steve work focused on construction [34]. For a Steve program to be memory safe entails two properties:

1. Within a processing stage, a header field shall not be accessed unless it is required or extracted.

2. For each node in the transition graph, the requirements of the node shall have been previously extracted.

In the first case, for a field to be used within a stage, the name must be in scope. In a decoder, a field name is in scope after an extract declaration for that field. Fields whose names are listed in an event handler’s requires specifier are required by the stage. The scope of these names is the body of the event handler. Similarly, field names listed in a flow table’s key and requires specifier are required by the stage, and the scope of these names is the body of initial and inserted flow entries for that table.

In the second case, if a stage requires a field, it must be extracted in all paths leading to that stage. For example, Figure 6.1 depicts an ill-formed pipeline where one path violates this property.

To check this, every vertex in the transition graph is associated with a set of productions and requirements. A production is a field that has been extracted or created by the vertex’s respective stage. Only decoding stages have productions. If a field were pushed onto the packet, that would also constitute a production; however,
Figure 6.1: An ill-formed pipeline. Table T1 requires fields A, B, and C. However, in the path from D1 to D3 to T1, field C has not been extracted.

this is not currently supported. Fields in a stage’s requirements are described in the first case.

Let $G$ be the transition graph representing the Steve pipeline. Let $P$ be the set of all paths from the root vertex (representing the starting decoder) to a vertex $v \in G$. For a path $p \in P$, let $S$ be the union of all productions of vertices in $p$. For $p$ to satisfy the requirements of $v$, the requirements of $v$ must be a subset of $S$. If any $p \in P$ does not satisfy the requirements of $v$, then the program is ill-formed. If all $v \in G$ have their requirements satisfied, then the program is well-formed, otherwise it is ill-formed.

To confirm that all vertices have their requirements satisfied, the Steve compiler uses depth-first traversal with backtracking to evaluate all possible paths in a pipeline graph (see Algorithm 6.1). This algorithm assumes that a graph is a DAG. As the algorithm traverses a path, it accumulates a set of productions at each vertex, then confirms that the vertex’s requirements are satisfied. Any vertex which fails
immediately indicates the program is ill-formed and further traversal along that path stops.

Once the algorithm reaches the end of a path successfully, it backtracks to previous vertices and explores other paths. When backtracking, the algorithm removes the productions of vertices that were left behind.

Algorithm 6.1 Depth-first traversal with backtracking used to check if the program is ill-formed because a field is accessed but not extracted.

Input: Let $G$ be the pipeline transition graph. Let $v$ be a vertex in $G$. Let $S$ be the accumulated set of productions.
Output: If the program is ill-formed because a field is accessed but not extracted.

function DFS($G, v, S$)
  v.visited = true
  difference = $S \setminus v$.productions
  $S = S \cup v$.productions
  if v.requirements $\subseteq S$ then
    for all $a$ in $G$.adjacentNodes($v$) do
      if a.visited == false then
        DFS($G, a, S$)
      else
        return ill-formed
    end for
  else
    return ill-formed
  end if
  v.visited = false \(\triangleright\) Reset the visited property so we can come down this vertex again in a different path.
  $S = S \setminus$ difference \(\triangleright\) Remove the productions of this vertex from the set of productions when backtracking.
end function

Theorem. The worst case complexity of this algorithm is $O(v!)$ where $v$ is the number of vertices.
**Complexity Proof.** Assume a DAG has \(v\) vertices. There are at most \(v - 1\) vertices adjacent to the root vertex. Each vertex adjacent to the root has at most \(v - 2\) vertices adjacent to it. This pattern continues until paths end in a sink. Thus the number of paths can be represented as \((v - 1)(v - 2)\ldots1\) or \((v - 1)!\). Therefore, the performance is \(O(v!)\).

Fortunately, this worst case performance is rarely ever achieved. Only the most dense pipeline graphs, where a vertex at each level is connected to every other possible vertex has this kind of performance. Packet processing graphs are typically closer to trees as branches only occur when determining what the next header is. These graphs are generally not very deep either as packets tend to have a very small number of headers. Additionally, it is acceptable to incur performance penalties at compile time to ensure safety at runtime.

**Termination Proof.** By definition, a DAG has a finite number of unique paths, therefore, the algorithm must terminate if it takes every unique path exactly once.

6.2 Why Objects of Layout Type can not Exist.

It was mentioned in an earlier chapter than objects of layout type may not exist. This is necessary for two reasons. First, Steve does not currently support creating new packets or pushing new headers. It can only process existing ones. Objects of layout type are not strictly necessary without this ability. Second, there are a number of concerns related to construction objects whose types are dynamically sized.
Allocating objects of dynamically-sized types (DST) on the stack produce a number of concerns which have not been solved. The only language which actually supports user-defined DSTs, Rust, only allows it under very limited circumstance [40]. Allocating DST objects on the heap is, of course, what dynamic allocation was made for, but it is expensive and does not help when reinterpreting existing packet memory. Allocating a single DST object on the stack is not actually a difficult thing to achieve. It is done by allocating memory by pushing the stack pointer forward using the intrinsic `alloca`. Now consider what happens when this DST object appears at the end of a class/record type. For example:

```c
struct Ipv4 {
    version : uint(4);
    ihl : uint(4);
    // ...
    // options has dynamically-sized type predicated on ihl.
    options : DST(ihl);
}

var x : Ipv4 = ...;
```

The `options` member has a length predicated on the value of another member in the object. In order to produce an object of type `Ipv4`, first the memory for the object must be allocated. Then the value of `ihl` has to be evaluated. Then the memory for `options` can be allocated using `alloca`. This is possible because `options` is the last member in the class. Though tricky, this is still possible and roughly what the Rust does.

However, now consider what happens when the DST member is *not* at the end of the class, but rather the middle. For example:
struct FooClass {
    x : int;
    y : DST(x);
    z : int;
}

Only x can be allocated at first. The value of x has to be evaluated and only then could y and z be allocated. It becomes impossible for the compiler to reason about the actual position of z in memory. This makes member access on z an extremely expensive runtime computation. Then consider that an object of type FooClass may also appear in another class, making it even more difficult to manage.

Because of these unclear issues, Steve has forgone allowing objects of DSTs. By extension, because layouts have to have DST fields, objects of layout type are also not allowed.

6.3 Decoder Anatomy

This section discusses the “anatomy” of a decoder. Specifically, it will describe the formalism of (offset, length) generation by the compiler.

6.3.1 Determining the Location and Length of Extracted Fields

An extract declaration produces an extraction by executing a set of instructions on a packet which saves the extraction’s location and length to the context’s binding environment. Information gathered from the field name and layout rule is used to calculate these two values. These instructions are completely opaque to the user and
are automatically generated by the compiler. The formalism for this code generation is described below.

First, let the field name have the form $E1.E2$ where $E1$ is the container layout and $E2$ is the contained field.

1. The length of the extracted field is calculated by a function $\text{len}(E2)$. The result of $\text{len}(E2)$ is the size of an object of $E2$’s type.

2. Let $\text{precede}(E1, E2)$ be a function that returns the sequence of all field declarations preceding $E2$ in $E1$.

3. The function used to calculate the relative offset of $E2$ is given as $\text{rel}(E1, E2)$ and is defined as:

   (a) If $E1$ identifies a layout, then $\text{rel}(E1, E2) = \sum_{x \in \text{precede}(E1, E2)} \text{len}(x)$.

   (b) If $E1$ is a field name, then let $E1$ have the form $E1’.E2’$.

   Then $\text{rel}(E1, E2) = \text{rel}(E1’, E2’) + \sum_{x \in \text{precede}(E2’, E2)} \text{len}(x)$

4. If $\text{rel}(E1, E2)$ results in a number greater than the length of the packet, then the packet is malformed and dropped.

5. Given a field name of the form $E1.E2$ and the current view index (see 3.2.4), $i$, the absolute offset of the field being extracted is $\text{abs}(E1, E2) = i + \text{rel}(E1, E2)$. If the result of $\text{abs}(E1, E2)$ is greater than the length of the packet, then the packet is malformed and dropped.
6. The location of an extracted field is its absolute offset, that is, the number of
bits (or bytes) it is away from the beginning of the packet. The pair stored by
the binding environment is \((\text{abs}(E_1, E_2), \text{len}(E_2))\).

Consider the following example. Two layouts, L and N, and a decoder, D, are
provided.

```cpp
layout L {  
a : uint;
b : uint;
c : N;
}
layout N {  
d : uint (8);
f : uint (16);
}
decoder D(L) {  
extrait L.a;
extrait L.c.f;
}
```

L has three fields: a, b, and c. N has two fields: d and f. Field L.c has layout
type N and is thus a nested layout. D decodes L and extracts L.a and L.c.f.

The first extract declaration is `extract L.a`. The type of a is `uint` (32 bit
unsigned integer by default). The result of `len(a)` is 32 bits or 4 bytes. The relative
offset of a in L is the sum of the lengths of all field declaration which precede it. In
this case, no fields precede it, thus its relative offset is 0.

The second extract declaration is `extract L.c.f`. This is the case of the
nested layout. The value of `len(f)` is 16 bits or 2 bytes. The relative offset of f is the
sum of the length of all field declaration which precede it in N (8 bits or 1 byte), plus
the relative offset of \( c \) in \( L \) (64 bits or 8 bytes). Thus, the relative offset for \( L.c.f \) is 
\[ 64 + 8 = 72 \text{ bits or 9 bytes} \]

Assuming an arbitrary example where the beginning of the current view for \( D \) is 112 bits past the beginning of the packet, the *absolute offset* for \( L.a \) would be 112 bits. The absolute offset for \( L.c.f \) would be \( 112 + 72 = 184 \) bits or 23 bytes.

### 6.3.2 Extracting the Same Field More Than Once

A common misunderstanding happens when an extract appears within a loop. A decoder is only ever looking at one header of a packet at once. Using an extract declaration with the same field name more than once in the same decoder will result in the same instance of that field being extracted multiple times, which is completely redundant. Though legal, this should be avoided in most cases. For example:

```c
while (x < 5)
    extract L1.f1;
```
CHAPTER VII
THE FREEFLOW RUNTIME AND STEVE

To load a Steve application, a switch must expose a consistent application binary interface (ABI) that the Steve program can call to access switch resources and optimized operations. To validate this approach, Steve targets a programmable, protocol independent, software data plane called Freeflow Virtual Machine (FFVM) [4]. FFVM virtualizes switch hardware as depicted in Figure 7.1. FFVM provides a runtime environment which exposes switch resources to the Steve application through an ABI.

Having a dedicated runtime environment provides Steve with two advantages. First, the ABI exposed by the runtime environment provides consistent resource access across all devices, meaning code generation differences between devices is minimal. Second, resource management is pushed into the data plane, making it completely opaque to the application. This way the application need not worry about discovering all resources on a device. The Steve application also provides an interface so that the switch may use it to configure its pipeline, decode packets, and access event handlers. The topic of this chapter will be describing these two interfaces.
Figure 7.1: The FFVM architecture. FFVM virtualizes the underlying switch hardware and switch OS. Steve programs are loaded by FFVM and instantiate the data plane’s pipeline. The Steve program also serves as the device’s controller.

7.1 Freeflow ABI

The Freeflow runtime environment exposes a set of system calls through compiled functions which allow applications to gain controlled access to switch resources. Table 7.1 summarizes the common usage of the library. Specifically, the ABI provides access to flow table allocation, flow entry manipulation, and ports.

7.2 Steve Applications

The Steve application is a dynamic link library (or shared object library). The Steve application provides an interface which a switch can use to modify its network function. Specifically, the switch uses this interface to configure its flow tables, access decoders, initiate pipeline processing on a context, or execute event handlers. Table 7.2 summarizes the application interface.
7.3 Steve Compilation

Steve compilation is a three stage process. In the first stage, high-level network-specific syntax is lowered into primitive code elements such as functions, function calls, variables, system intrinsics, etc. In the second stage, Steve is compiled into LLVM IR [5]. The LLVM IR is linked against the Steve runtime library which contains the context data structure. The context data structure is kept in an independent library so that different context formats may be easily swapped out depending on the target architecture. In the last stage, the LLVM IR is compiled by the LLVM optimizing compiler into assembly code which is assembled into the DLL.

7.4 Application Lifetime

When FFVM loads a Steve application library, it calls `ff_load` which initializes flow tables. Once loading is complete, FFVM calls the `ff_process` function to dispatch a context to the first decoder. FFVM will use `ff_start` and `ff_stop` to start and stop the delivery of events to the application. When FFVM wants to remove the application, it calls `ff_unload` which releases all resources requested by the application.
Table 7.1: The Freeflow ABI that may be used by applications. The ABI and its function implementations are subject to change, so specific parameters and return types are not given.

<table>
<thead>
<tr>
<th>Interface Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ff_output(context, portid)</code></td>
<td>Creates a copy of the packet stored with the context and forwards it to the port with the given id.</td>
</tr>
<tr>
<td><code>ff_drop(context)</code></td>
<td></td>
</tr>
<tr>
<td><code>ff_goto_table(context, table, key)</code></td>
<td>Sends a context to be matched against a given table. The application is responsible for telling the data plane which fields must be extracted from the context and turned into a query key.</td>
</tr>
<tr>
<td><code>ff_create_table(dataplane, key_width, size, type)</code></td>
<td>Instructs a data plane instance to construct a flow table and return a handle to that flow table. The application is responsible for informing the data plane of the key width, the maximum number of flow entries (size), and the match pattern (type).</td>
</tr>
<tr>
<td><code>ff_insert_flow(table, actions, key, info)</code></td>
<td>Allows for adding initial flow entries, new flow entries, and miss case flow entries to a given table.</td>
</tr>
<tr>
<td><code>ff_insert_miss(table, actions, info)</code></td>
<td></td>
</tr>
<tr>
<td><code>ff_remove_flow(table, key)</code></td>
<td>Allows for deleting flow entries with certain match fields, or for removing the miss case.</td>
</tr>
<tr>
<td><code>ff_remove_miss(table)</code></td>
<td></td>
</tr>
<tr>
<td><code>ff_raise_event(context, handler)</code></td>
<td>Requests that the runtime queues up a context and an event handler on the controller port. The controller then executes the event handlers as it receives the contexts. Execution may not be immediate.</td>
</tr>
<tr>
<td><code>ff_port_id_is_up(portid)</code></td>
<td></td>
</tr>
<tr>
<td><code>ff_port_id_is_down(portid)</code></td>
<td>Returns true or false depending on whether a port is up or down.</td>
</tr>
</tbody>
</table>
Table 7.2: The application interface. All Freeflow loadable applications must provide this interface.

<table>
<thead>
<tr>
<th>Interface Functions</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff_load(dataplane)</td>
<td>This function is used for configuration when an application gets loaded. Freeflow passes a data plane handle to this function. This function then calls the Freeflow ABI to set up flow tables and flow entries.</td>
</tr>
<tr>
<td>ff_unload(dataplane)</td>
<td>This function releases all dataplane resources allocated by the application.</td>
</tr>
<tr>
<td>ff_process(context)</td>
<td>Freeflow sends contexts through this function to start pipeline processing. The application’s logic will dictate which decoders and tables are used. Once the application finishes execution, Freeflow executes egress processing described in Section 3.2.12.</td>
</tr>
</tbody>
</table>

Event Handlers

<table>
<thead>
<tr>
<th>Event Handlers</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff_start(dataplane)</td>
<td>These are used to start and stop the delivery of events.</td>
</tr>
<tr>
<td>ff_stop(dataplane)</td>
<td>Event handlers may vary in name, but all accept a context as a parameter. Whenever an event is raised, an event handler with the same name is executed.</td>
</tr>
</tbody>
</table>
CHAPTER VIII
EXPERIMENTS

This thesis is not performance based. However, to verify some questions and hypotheses about elements of the language design, some performance experiments were performed. These experiments were performed using Fakeflow, a Freeflow data plane emulator. Fakeflow is largely equivalent to a single-threaded implementation of Freeflow with the exception of ports. Fakeflow removes the overhead of sending and receiving from ports. Instead, Fakeflow reads packets from pcap files to emulate receiving from ports and forwarded packets do not get sent anywhere (equivalent to simply dropping the packet). This approach is taken because the speed at which the data plane outputs packets varies largely with implementation. The goal of this chapter is to focus only on the performance of the application, not the speed of switch resources. Specifically, two questions are evaluated.

1. Do partial decodes produce measurable gains over full decodes? Steve was designed to support partial decodes, so for the design choice to be valid, this must be true.
2. What is the performance of each individual action or operation on a general purpose CPU? Knowing this allows the programmer to make certain optimization choices when writing code.

8.1 Data Sets

Tests were performed using three pcap samples described in Table 8.1. The samples are relatively diverse, having varying packet sizes, packet count, and protocols. Samples 1 and 2 were taken from Tcpreplay’s sample captures \(^1\) used for testing NetFlow. Sample 3 was taken from Netresec \(^2\) which contains real industrial network traffic passing through ICS labs.

Table 8.1: The three pcap test cases used for experiments.

<table>
<thead>
<tr>
<th>#</th>
<th>Pcap sample</th>
<th>Packet Count</th>
<th>Average Packet Size</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smallFlows</td>
<td>14,261</td>
<td>646 bytes</td>
<td>A synthetic capture using various different protocols.</td>
</tr>
<tr>
<td>2</td>
<td>bigFlows</td>
<td>791,615</td>
<td>449 bytes</td>
<td>Real network traffic from a busy private network’s Internet access point.</td>
</tr>
<tr>
<td>3</td>
<td>4SICS</td>
<td>2,274,747</td>
<td>76 bytes</td>
<td>Real industrial network traffic from ICS labs.</td>
</tr>
</tbody>
</table>

\(^1\)http://tcpreplay.appneta.com/wiki/captures.html
\(^2\)http://www.netresec.com/?page=PCAP4SICS
8.2 Partial vs. Full Decodes

Steve proposed that partial decodes produced measurable gains over full decodes. To demonstrate this, an experiment was run using the stateless firewall application. The original version had partial decodes of ethernet, TCP, and UDP headers. A duplicate was written with full decodes of all headers.

Each sample is run through a certain number of iterations, so that approximately 90 million packets from each sample are sent through. The timeout property for flow entries in all examples were removed so flow entries last indefinitely. Tests were run on an Intel(R) Core(TM) i3-2130 CPU with a speed of 3.40GHz. The Freeflow emulator was configured to have a fixed number of ports. The ingress port for each packet was randomly assigned amongst these ports. For each experiment, there were five ingress ports. Table 8.2 summarizes these results. The table also includes the percentage of TCP and UDP packets per sample. The full decoding firewall extracts four additional fields on UDP packets and nine additional fields on TCP packets.

Table 8.2: Comparing firewall performance (in Mpps) with partial header decodes versus full header decodes.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Iter.</th>
<th>Partial (Mpps)</th>
<th>Full (Mpps)</th>
<th>% TCP</th>
<th>% UDP</th>
<th>Approx. Diff. (# packets)</th>
<th>Percentage Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6380</td>
<td>1.64</td>
<td>1.56</td>
<td>96.12%</td>
<td>3.67%</td>
<td>80,000</td>
<td>5.00%</td>
</tr>
<tr>
<td>2</td>
<td>114</td>
<td>1.75</td>
<td>1.63</td>
<td>80.22%</td>
<td>19.45%</td>
<td>120,000</td>
<td>7.10%</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>2.06</td>
<td>1.91</td>
<td>95.25%</td>
<td>3.68%</td>
<td>150,000</td>
<td>7.56%</td>
</tr>
</tbody>
</table>
A partial decode produces a non-negligible difference in performance with the average percentage difference between the three samples being about 6.55%. This is only for a simple three header packet. For more complex header structures, gains from partial decodes would become more and more substantial.

8.3 Performance of Operations

This section presents the typical performance of certain common operations performed by a Steve pipeline. Each operation is executed approximately 10 million times per experiment. This experiment is repeated ten times and the timings are given as an average. Table 8.3 summarizes the timed performance of each action. The output action is intentionally excluded as the results vary based on a number of factors including Freeflow implementation, threading models, and specialized forwarding hardware.

To reiterate the conclusion of Section 8.2, the average time it takes to decode a field is about 23 nanoseconds (ns). This translates to approximately 0.23 seconds of processing time per extra field extracted every 10 million packets. This is a non-negligible difference as many networking devices do expect such heavy traffic.

As mentioned in earlier sections, inserting and removing flow entries are the most expensive operations a pipeline can perform. They do notably bottleneck the pipeline and are best performed asynchronously. It is also worth noting that writing actions to an action list is more expensive than one might anticipate (being only
slightly faster than table matching). This may be a result of the current implementation dynamically allocating memory to store the action.

Table 8.3: Average wall clock time for executing certain operations. Output action has been excluded as it varies with the threading model implementation of Freeflow.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (nanoseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goto action + Table matching</td>
<td>110.49</td>
</tr>
<tr>
<td>Insert flow entry action</td>
<td>418.66</td>
</tr>
<tr>
<td>Remove flow entry action</td>
<td>273.53</td>
</tr>
<tr>
<td>Field decode</td>
<td>23.22</td>
</tr>
<tr>
<td>Field access / read</td>
<td>35.02</td>
</tr>
<tr>
<td>Field write / set action</td>
<td>29.34</td>
</tr>
<tr>
<td>Write set action</td>
<td>140.76</td>
</tr>
<tr>
<td>Write output action</td>
<td>108.92</td>
</tr>
</tbody>
</table>

The goto action (which also invokes table matching) is also relatively expensive compared to other operations. It is for this reason that users should strive to reduce table matches as much as possible. The developers of POF came to a similar conclusion [13]. They suggest using as few tables as possible. Multiple tables can be combined into a single table. When appropriate, a user should also consider replacing a table with a conditional statement.
CHAPTER IX
CONCLUSION AND FUTURE WORK

The goal of this research was to present a safe and efficient language for writing network applications for SDN switches. Though other languages already exist, Steve sought to improve upon them. First, Steve provides strict compile-time safety guarantees of user-defined packet processing pipelines. Steve’s type and constraints system make it almost impossible to write a network application which invokes undefined behavior. Secondly, the Steve language expands past the basic pipeline specification of most languages and allows the programmer to define protocol independent control plane event handling as well. With that being said, there are still issues we have yet to explore.

9.1 Future Work

Steve is far from a complete and mature language. It is still missing key components that other languages of its kind already support. Dynamically-sized fields, group tables, alternative table matching patterns (prefix, wildcard, etc), pushing and popping headers, packet construction, etc. On top of that, we have yet to determine a programming model for distributed control.
9.1.1 Language Use Cases

As we are not network programming specialists, we were never certain of which language features were needed to support rich network applications. We chose to use OpenFlow [1] as our guidepost of what was, at minimum, required. From there, we expanded on it from other languages such as POF [12, 11, 13] and P4 [12, 10, 8]. Steve may expand as the need for new use cases grow.

Pushing and popping headers as well as packet construction are the most interesting. This would allow Steve to become more than purely a packet processing language. The Steve programmer could hypothetically create new packets as a reactive response to certain network traffic.

Device introspection, that is, the ability of the application to learn things about the switch is missing. It may be useful for an application to search for a specific port, or to learn about the device’s MAC and IP addresses. Deep packet introspection is similarly useful. Most SDN languages only focus on headers, but being able to look into a payload allows certain security applications to be written on the switch.

9.1.2 Portability

Portability is another major issue closely related to use cases. We do not own the SDN hardware capable of running Steve due to cost. Tests were performed on general purpose processors instead. We are also unaware of how many SDN switches are actually fully capable of all the Steve features. At the time of writing, these
devices are rather exotic and lack standardization. This is why we choose to target the interoperable ABI from the Freeflow rather than compiling for specific switch instructions.

First the virtual machine, Freeflow [4] or some alternative software, would have to be ported to the target device to expose the ABI which Steve depends on. Then the Steve compiler would have to be rewritten to target the appropriate processor architecture. We suspect this is rare because LLVM already supports the most common ones.

9.1.3 Runtime ABI

The runtime ABI is designed to alleviate issues with compiling to other platforms. As long as there is the ABI exposes functions, Steve can be compiled to target them. The advantage is that the runtime environment worries about managing hardware acceleration, optimization, etc., rather than the compiled Steve application. In future work, we believe it is possible to provide optimized resources (such as TCAM) to the application opaquely through the runtime ABI.

9.1.4 Language Safety

We have proven how Steve enforces pipeline safety, but because Steve focuses on being safe over everything else, the language enforces very strict rules about pipeline composition. However, some of these rules can be weakened depending on use cases. Specifically, the constraint that pipelines cannot have cycles can be relaxed. It is possible to loosen this restriction so that pipelines may have cycles, but only between
decoders. This is useful for recursively decoding an indeterminate number of headers of the same kind (like VLAN). This would not cause an infinite cycle because the view would eventually shift off the end of the packet causing it to be dropped.

9.1.5 Distributed Control

It is possible that, rather than compiling one binary application, it is possible for the Steve compiler to produce two: one containing data plane entities, and another containing control plane event handlers. The binary containing event handlers could be loaded by another machine and act as distributed controller. The message format could potentially be the context data structure wrapped by network protocols. Alternatively, since we can process any protocol, including OpenFlow, it is possible to process OpenFlow messages and locally and respond accordingly.

9.2 Conclusion

Being able to program every component of a network switch is a powerful thing. Just by changing out a program, the entire behavior of the device can change. This is an incredibly powerful thing. It allows enterprises to quickly reconfigure their entire network setup at little to no cost. It also expands the possibilities of how the Internet may one day evolve. As SDN aims to make switches more like general purpose computers, the Internet may too evolve into something better. With that being said, SDN devices are far from being commonplace. As it stands, SDN trades speed for flexibility and lacks standardization, making enterprises hesitant to adopt it.
Steve is just an infant step towards this potential future. This system addresses what a protocol independent, safe, and efficient SDN programming environment looks like. This research provides a solid foundation of language features that must exist to implement network applications that can hopefully expand in the future.
BIBLIOGRAPHY


[19] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic.


APPENDIX

THE STEVE REFERENCE GUIDE

This chapter serves as a reference manual for Steve programmers. It should be used as a reference rather than being read through. It provides detailed descriptions of Steve language semantics: how features work, type checking, grammar, conventional and illegal uses, etc. It also describes how pipeline safety guarantees are enforced, what constitutes a legal pipeline, and potential pitfalls a programmer might fall into when writing a pipeline. There is an expectation that the user has read the Tutorial and has a basic understanding of Steve.

Grammar in this section is given in Backus-Naur Form (BNF). All non-terminals are given between < >. All terminals appear in mono-spaced font. All optional grammar items are given between [ ]. Grammar items followed by a + indicate that 0 or more of that item may be present. All identifiers are suffixed with \texttt{-id} and all names given to declarations are suffixed with \texttt{-name}.

A.1 Identifiers

An \textit{identifier} is an arbitrarily long sequence of characters. Supported characters include uppercase Latin letters (A - Z), lowercase Latin letters (a - z), digits (0 -
Identifiers are case sensitive. The grammar of identifiers is as follows:

\[
\text{⟨identifier⟩} ::= \langle \text{letter} \rangle \langle \text{identifier-characters} \rangle + \\
\text{⟨identifier-character⟩} ::= \langle \text{letter} \rangle \\
| \langle \text{digit} \rangle \\
| _ \\
\text{⟨letter⟩} ::= \text{A} | \text{B} | \text{C} | \text{D} | \text{E} | \text{F} | \text{G} | \text{H} | \text{I} | \text{J} | \text{K} | \text{L} | \text{M} | \text{N} | \text{O} | \text{P} | \text{Q} | \text{R} | \text{S} | \text{T} | \text{U} | \\
| \text{V} | \text{W} | \text{X} | \text{Y} | \text{Z} | \text{a} | \text{b} | \text{c} | \text{d} | \text{e} | \text{f} | \text{g} | \text{h} | \text{i} | \text{j} | \text{k} | \text{l} | \text{m} | \text{n} | \text{o} | \text{p} | \text{q} | \text{r} | \text{s} | \text{t} | \text{u} | \text{v} | \text{w} | \text{x} | \text{y} | \text{z} \\
\text{⟨digit⟩} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

Identifiers are subject to the following limitations:

- Identifiers which are keywords cannot be used for other purposes (see Section A.1.1).

- Identifiers beginning with double underscores (__), underscores followed by a capital letter (ex. _F) or an underscore followed by a capital letter (ex. _F) are reserved by the compiler for internal identifiers.

Identifiers can be used as names for entities. An entity is a value, object, reference, function, layout, layout field, decoder, table, flow entry, port, event, or extraction. In the Steve grammar, identifiers being used as names for a kind of declarations shall have the form ⟨kind-name⟩, where kind shall be the declaration kind. For example, a decoder name would be represented in the grammar as ⟨decoder-name⟩.

Identifiers that refer to a variable, function, decoder, table, or port can be used as an expression. In this case, the identifier becomes an identifier expression (see A.10.2). In the Steve grammar, identifier expressions shall have the form ⟨kind-id⟩,
where \textit{kind} shall be the declaration kind. For example, an identifier to a variable would be represented in the grammar as \texttt{(variable-id)}.

A.1.1 Keywords

A number of identifiers in Steve are reserved as \textit{keywords}. The meaning and semantics of these identifiers cannot be changed. A list of Steve keywords can be found in Table A.1.

<table>
<thead>
<tr>
<th>bool</th>
<th>break</th>
<th>char</th>
<th>continue</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>else</td>
<td>foreign</td>
<td>int</td>
<td>uint</td>
</tr>
<tr>
<td>return</td>
<td>struct</td>
<td>this</td>
<td>var</td>
<td>while</td>
</tr>
<tr>
<td>match</td>
<td>case</td>
<td>layout</td>
<td>decoder</td>
<td>decode</td>
</tr>
<tr>
<td>start</td>
<td>extract</td>
<td>as</td>
<td>exact_table</td>
<td>requires</td>
</tr>
<tr>
<td>miss</td>
<td>Port</td>
<td>goto</td>
<td>output</td>
<td>write</td>
</tr>
<tr>
<td>drop</td>
<td>flood</td>
<td>clear</td>
<td>set</td>
<td>insert</td>
</tr>
<tr>
<td>remove</td>
<td>into</td>
<td>from</td>
<td>event</td>
<td>raise</td>
</tr>
<tr>
<td>in_port</td>
<td>in_phys_port</td>
<td>all</td>
<td>controller</td>
<td>reflow</td>
</tr>
<tr>
<td>advance</td>
<td>egress</td>
<td>struct</td>
<td>char</td>
<td></td>
</tr>
</tbody>
</table>

A.2 Field Names

\textit{Field names} are special, \textit{qualified} names which refer to field declarations made within a layout declaration (see A.8.1). Field names have the form:

\[
\langle \text{field-name} \rangle ::= \langle \text{layout-id} \rangle . \langle \text{field-id} \rangle \\
| \langle \text{field-name} \rangle . \langle \text{field-id} \rangle
\]
Given a field name of the form \( E_1.E_2 \), if \( E_1 \) is a valid identifier to a layout declaration, then \( E_2 \) must be a field of \( E_1 \). In other words, \( E_2 \) must be found in the scope of \( E_1 \) using a qualified name lookup (see A.3.9).

If \( E_1 \) is a valid field name, and the referred to field has type \( T \), then \( T \) must be layout type (see A.6.8) declared by a layout declaration (see A.8.1), \( D \). \( E_2 \) must identify a field of \( D \). In other words, \( E_2 \) must be found in the scope of \( D \) using qualified name lookup.

\( E_1 \) is referred to as the container and \( E_2 \) is referred to as the contained field. The root layout identifier is used to mean the leftmost identifier in a field name. For example, in \( \text{l1.f1.f2.f3} \), \text{l1} is the root layout identifier.

Field names used as expressions become field access expressions (see A.10.3). Field access expressions may be used to refer to the value of the last extraction with that field name.

A.3 Scope

Steve scope semantics work like all C-based block-structured languages (including C++, Java, C#, etc). Declarations are used throughout program text to introduce names which refer to an entity (see A.1).

A name is only valid within parts of program text called the scope of that name. A particular name is only considered valid if an entity with that name can be found using an unqualified name lookup (see A.3.8).
A.3.1 Global Scope

The outermost part of program text where declarations can be made is known as *global scope*. All declarations made at global scope are said to be *global declarations* and their names are said to be *global names*.

Global names are valid at any point in the program. Steve does not require forward declarations. Two declarations of the same name that are not overloaded functions shall not be made at global scope.

A.3.2 Block Scope

Blocks are portions of program text which can have their own local declarations. The beginning of a block is delimited by the left-brace (\{\}) and the end of a block is delimited by the right-brace (\}). Blocks define a block scope which limits the scope of declarations made within them. A declaration made within a block is in *block scope* and is *local* to that block.

The scope of a name made within a block begins at the point of declaration and ends at the final left-brace of the block. It is illegal to use the name of a declaration in local scope at a textual position before its declaration. Two different declarations of the same name cannot be made inside the same scope.

Scopes can be nested. In this case, the inner scope is said to be the *enclosed* scope, and the outer scope is said to be the *enclosing* scope. The same name can be declared in the enclosing scope, and again in one or more enclosed scopes. The same is true for further nested scopes within the enclosed scope. If this happens,
the scope of the outer declaration is its typical scope excluding program text of the
enclosed scope. Unqualified name lookup (see A.3.8) shall be used to unambiguously
determine which declaration the name refers to.

For example, in the following example, the name i is declared twice. The
scope of the first i is global scope and includes the entire example excluding the
block between the first left-brace ({}) and the closing right-brace (}). The scope of
the second i begins at its declaration and ends at the the closing right-brace (})

```plaintext
var i : int = 0;
def f() -> int {
  var i : int = 1;
  var j : int = 2 + i;
  return j; // Result here shall be 3.
}
```

The body of if, while, and match statements all implicitly introduce a block.

A.3.3 Function Scope

The scope of a parameter declared in a <function-declaration> (see A.8.9) is the
function’s <block-statement>.

A.3.4 Table Scope

A name is said to be in table scope if its scope is within the <table-initializer>
of a <table-declaration> (see A.8.5). The scope of field names declared by a
<key-declaration> within <key-declaration-sequence> and field names listed in
the <requires-specifier> is the <table-initializer>.
The field name refers to the last extraction made with that field name. This entity may vary during runtime depending on the path through the pipeline which a packet took. Because of memory safety (see 6.1.3), it is always guaranteed that an entity with that name exists.

A.3.5 Flow Scope

A name is said to be in flow entry scope if its scope is within the \textit{flow-body} of a \textit{flow-declaration} (see A.8.5). If a \textit{flow-declaration} appears within a \textit{table-initializer}, then all field names whose scope is \textit{table-initializer} are valid within the flow entry’s body.

If a flow entry is being inserted into a table using the \textit{insert-action} (see A.12.7), then a flow declaration is implicitly generated representing that flow entry. The \textit{flow-body} of that declaration is treated as if it were within that table’s \textit{table-initializer}. This ensures that the field names valid within that table are valid within the inserted flow entry’s body.

Names found in the \textit{properties-block} of an inserted flow entry follow regular scope semantics. That is, all names which are normally valid at the point where the \textit{properties-block} appears in the program text are valid inside the \textit{properties-block}. However, these names may not be valid for the \textit{flow-body} in the case of an \textit{insert-action}.

In the following example, the name \textit{x} is valid in the properties block, but not in the body of the flow.
event e1 {
    var x : int = 100;
    insert into t1
    { x } -> // 'x' is valid here.
    [ timeout = x ] // 'x' is also valid here.
    {
        // Error: The name 'x' is not valid here.
        set eth.type = x + 1;
        // However, the field name 'eth.type' is valid
        // because the flow body is treated as if nested
        // inside the scope of table 't1', whose key declaration
        // 'eth.type' makes the field name valid.
        flood;
    }
}

exact_table t1(eth.type) { }
• After the dot-operator (.) applied to either a field name (see A.2) or a field access expression (see A.10.3).

A.3.8 Unqualified Name Lookup

Unqualified name lookup attempts to find the corresponding declaration for a name being used. Unqualified name lookup begins at the innermost scope, before the name is used and works outward toward enclosing scopes. If the declaration is not found in any enclosing scopes, global scope is searched. The innermost declaration (e.g. the first one found) with that name is considered the corresponding declaration.

A name must be declared before being used. Any attempts to use a name before its declaration shall result in a failed unqualified name lookup. If unqualified lookup fails to find a corresponding declaration, the a compiler error is produced.

If the name refers to one or more function declarations, and is being used as a function call, an overload set is associated with the name. If this is the case, argument dependent lookup determines which declaration is being referred to. The function declaration chosen shall be the one whose parameter types match the argument types used in the function call.

Global names used at any point are considered valid regardless of the order with which they are declared in the program text. For example, the following usage of the name i in function foo refers to a variable declaration made after the function declaration. This is considered valid.

```python
// The name 'i' is used even though it is declared later // in global scope.
def foo() -> int { return 3 + i; }
```
// The name 'i' is declared here.
var i : int = 0;

A.3.9 Qualified Name Lookup

Qualified name lookup attempts to find the corresponding declaration for a name in a given scope. The search is done only on the given scope before the usage of the name and does not expand outward to enclosing scopes.

Qualified name lookup is used for looking up names following the dot-operator in field names (see A.2) and field access expressions (see A.10.3).

If qualified lookup fails to find a corresponding declaration, the result is a compiler error.

A.4 Objects

An object in a Steve program is an area of memory that has size, lifetime, type, and value. An object may also be given a name. Variables, extractions, ports, tables, and temporaries are objects. Objects are created in Steve by variable, extract, port, and table declarations (see A.8.8, A.8.3, A.8.7, A.8.5). Objects may be created where temporary values are required. For example, the evaluation of expressions such as addition (see A.10.17) produce temporary objects.

A.4.1 Storage Duration and Lifetime

The storage duration of an object describes the point where an object’s memory is allocated and the point where its memory is deallocated.
• **Automatic.** Automatic storage duration says that an object is allocated at the beginning of the code block where it is declared and deallocated at the end of a code block. Global entities (such as tables, ports, and global variables) and local variables have automatic storage.

• **Pipeline automatic.** Pipeline automatic storage duration is a property of the context object (see A.5). Upon packet ingress, the context is allocated, upon egress the context is deallocated. The memory associated with extracted fields, which are part of the packet, exist before they have actually been extracted, and thus have the same storage duration.

  The *lifetime* of an object is less than or equal to its storage duration.

• **Automatic.** Automatic lifetime begins at the point where an object is *initialized* and ends at the point where an object is de-initialized (which is currently equivalent to the end of the block).

• **Pipeline automatic.** Pipeline automatic lifetime is equivalent to pipeline automatic storage duration for contexts. The lifetime of an extracted field can be thought of as being from the point where its position is found (decoded) to the point where the context’s storage duration ends.

A.5  Context Data Structure

The storage duration and lifetime of a context is said to be *pipeline automatic* (see A.4.1). Management of the context’s memory is exclusively and automatically han-
handled by internal mechanics of the runtime system. This object is used exclusively by
the internals of the program to keep track of information about a packet. It is also
important to note that the packet itself is also a part of the context. Any point where
"the context" is referred to, it is implied to mean both the packet and context.

A user is never given direct access to a context. Context objects are implicit,
yet invisible, within all pipeline processing stages (decoders, tables, and events).
They serve to carry data as input and output between stages. Certain expressions,
declarations, and actions might access, modify, or otherwise affect the data of the
context in limited and well-defined ways.

- Expressions which access a context are: field access expressions (see A.10.3),
ingress port expression, and ingress physical port expression (see A.10.4).

- Declarations which modify a context are: extract declarations (see A.8.3) and
rebinding declarations (see A.8.4).

- All actions (see A.12) will affect the context in some way.

A.6 Type

Types are a way of describing objects, references, expressions, fields, and functions.
Types can describe size, structure, and limitations on possible values and legal op-
erations for these entities. By giving types to these things, it allows the compiler
to reason about limitations on where terms may appear. Steve is a strongly-typed
language. Therefore, the types of all things are known during compile time.
Steve supports scalar types (see A.6.1), reference types (see A.6.5), layout type (see A.6.8), port type (see A.6.9), and function type (see A.6.10). Types may appear as terms in language to specify the type of variables, parameters, return values, and layout fields. At other points, the type of an entity may be implicit. For example, the type of an additive expression (see A.10.17) is implicitly integer. Types have the form:

\[
\langle \text{type} \rangle ::= \langle \text{scalar-type} \rangle \\
| \langle \text{reference-type} \rangle \\
| \langle \text{function-type} \rangle \\
| \langle \text{layout-type} \rangle \\
| \langle \text{port-type} \rangle
\]

A.6.1 Scalar Type

Objects of scalar type represent exactly one value. They are not composed of multiple sub-objects. Steve supports the scalar types boolean, integer, and character. Scalar types have the form:

\[
\langle \text{scalar-type} \rangle ::= \langle \text{boolean-type} \rangle \\
| \langle \text{integer-type} \rangle \\
| \langle \text{character-type} \rangle
\]

A.6.2 Boolean Type

Objects of boolean type may only have the values \text{true} and \text{false}. Objects of boolean type have a size of 8 bits. They have the form:

\[
\langle \text{boolean-type} \rangle ::= \text{bool}
\]
A.6.3 Integer Type

An integer type describes the size, precision, and signed/unsigned-ness of a given integer value. They have the form:

\[
\langle \text{integer-type} \rangle ::= \langle \text{signed-integer-type} \rangle \\
| \langle \text{unsigned-integer-type} \rangle
\]

\[
\langle \text{signed-integer-type} \rangle ::= \text{int} [ \langle \text{precision-specifier} \rangle ]
\]

\[
\langle \text{unsigned-integer-type} \rangle ::= \text{uint} [ \langle \text{precision-specifier} \rangle ]
\]

\[
\langle \text{precision-specifier} \rangle ::= (\langle \text{integer-literal} \rangle)
\]

The precision of an integer is given by \langle \text{precision-specifier} \rangle. Precision is not always equivalent to size. The size of an integer is always the next highest multiple of 8. If \langle \text{precision-specifier} \rangle is not given, the default precision and effective size shall be 32. For signed integers, the range of integers that can be represented is from \(-2^{N-1}\) to \(2^{N-1} - 1\), where \(N\) is the precision. For unsigned integers, the range of integers that can be represent is 0 to \(2^N - 1\).

A.6.4 Character Type

Single characters have character type. Objects of this type have a size of 8 bits. It has the form:

\[
\langle \text{boolean-type} \rangle ::= \text{char}
\]

A.6.5 Reference Type

Objects and expressions of reference type represent an address to an object. Reference types have the form:

\[
\langle \text{reference-type} \rangle ::= \langle \text{type} \rangle &
\]
The type T& is read as "reference-to-T," where T is the type of the referred to object.

Identifier expressions referring to variables have reference type. Field access expressions refer to extractions and thus have reference type.

For example, in the following, the variable x has type int and the identifier expression x would have type T&.

```plaintext
x; // This identifier expression has type ref T
// Used as a value, reference-to-value conversion
// is applied to x
var y : int = x + 10;
```

When an expression of type T& appears on the right-hand side of an assignment statement (see A.11.4), a reference-to-value conversion (see A.7.1) is applied to recover the value of the object.

Variables of reference type may be created. Assignment to that variable modifies the original object as well. For example:

```plaintext
var y : int& = x; // y is a reference to x.
y = 5; // This modifies x as well, so now both are 5.
```

Here, any modifications to y have the same affect on x, because y only stores an address to the original object.

A.6.6 Array Type

An array type, T[N], describes a region of contiguous memory containing N number of objects of type T. Array types have the form:

```plaintext
⟨array-type⟩ ::= ⟨type⟩ [⟨expression⟩]
```
The type of \( \textit{expression} \) shall be integer. The evaluated value shall be a positive integer, otherwise behavior is undefined.

### A.6.7 Block Type

A block type describes a region of contiguous memory containing an unknown number of objects of type \( T \). Block type has the form:

\[
\langle \text{block-type} \rangle ::= \langle \text{type} \rangle [\ ]
\]

Block type may be used when the length of an array is unknown during compile time. Block type is most frequently given to variables used to hold strings. For example:

```plaintext
var str : char[] = "Hello, world";
```

### A.6.8 Layout Type

Layout types are declared via layout declarations (see A.8.1). Layout types have the form:

\[
\langle \text{layout-type} \rangle ::= \langle \text{layout-id} \rangle
\]

Objects of layout type shall not exist. The \( \langle \text{layout-type} \rangle \) may only appear as the type of a field declaration within a layout declaration (see A.8.1). It may not appear in variable declarations or function declarations as such:

```plaintext
layout L1 { f1 : uint; f2 : uint(16); }
var x : L1 = 0; // Invalid
// Invalid parameter type and return type.
def foo(y : L1) -> L1 { ... }
```
The size of a layout is equal to the sum of the sizes of each of its field types. This is only knowable at compile time if the layout contains only scalar typed fields. For example, the layout L1 has a size of 48 bits or 6 bytes.

A.6.9 Port Type

All port declarations have port type. Port type is not a type that appears explicitly in the grammar. Variables, parameters, and fields may not have port type.

In certain cases, namely ordering and equality operators (see A.10.19), an object of port type may be converted to an object of integer type. The value of the converted port object is equal to its system assigned integer identifier. This allows for the comparison of ports to integers as follows:

```c
Port p1;
// Comparison against 0 checks if the port is valid.
// The 0 port is the invalid port.
if (p1 == 0) { ... }     
```

A.6.10 Function Type

Functions have function type. Function type describes the type of each parameter and the return type of the function. Function type can be written and has the form:

```c
⟨function-type⟩ ::= ⟨⟨type-sequence⟩⟩ ↦ ⟨⟨type⟩⟩
```

A function of type \((T_1, T_2, \ldots, T_n) \Rightarrow T\) has parameters of type \(T_1, T_2, \ldots, T_n\), respectively, and a return type \(T\).
A.7 Conversions

At certain points, the type of an expression may be implicitly converted to another type. A *conversion sequence* applies conversions in turn until the original type has been converted to another target type.

A.7.1 Reference to Value Conversion

Given an assignment statement (see A.11.4) \( E_1 = E_2 \), an expression of type \( T\& \) which can appear as \( E_1 \) or in \( E_2 \) is traditionally called an *lvalue* in C/C++ [41]. Lvalues are objects which persist past a single expression. These are typically identifiers which refer to non-temporary objects. Expressions which may only appear in \( E_2 \) are known as *rvalues*. Rvalues do not persist past the expression which creates them such as: literals, arithmetic expressions, comparison expressions, etc.

When an lvalue of type \( T\& \) appears in \( E_2 \), it must go through implicit reference to value conversion (also known as lvalue to rvalue conversion). The reference is replaced with the object it refers.

```plaintext
var x : uint = 10;
// The reference to value conversion is implicitly applied on the identifier expression
// 'x' here.
var y : uint = x + 5;
```

A.7.2 Integer Conversions

Steve will convert the precision and signed/unsigned-ness of integers where necessary.

If an expression \( E_1 \) of type \( T_1 \) must be converted to an integer of type \( T_2 \), the following conversions each get applied in order.
1. If T1 is a signed integer and T2 is an unsigned integer, then the type of E1 is converted to an unsigned integer type with the same precision as T1 (the original precision).

2. If the precision of T1 is less than the precision of T2, then the type of E1 is converted (promoted) to an integer type with the same precision as T2. The extra high-order bits are zero-filled if both are unsigned or sign-extended if both are signed.

3. If the precision of T1 is greater than the precision of T2, then the type of E1 is converted (demoted) to an integer type with the same precision as T2. The extra high-order bits are truncated.

A.7.3 Arithmetic Conversions

In an arithmetic expression (see A.10.16, A.10.17, A.10.18, A.10.20), the resulting integer type is decided based on the rules below. Given two operands, E1 of type T1 and E2 of type T2, the resulting integer type:

1. has the same precision as the largest precision between E1 and E2.

2. is unsigned if either T1 or T2 are unsigned. Otherwise, it is signed.

A.7.4 Port to Integer Conversions

In some cases, namely comparison and equality operators, a port expression (see A.10.4) may be converted to have integer type. A port to integer conversion takes a port object and interprets its value to an integer value. The integer value is equal
to the port object’s system assigned ID number. Port objects which have not been initialized or are invalid have an integer value 0.

For example, in the following case, a port to integer conversion is applied to expression p1 to allow it to be compared with 0 (note that a reference-to-value conversion is actually applied first since p1 is an identifier expression with reference-to-port type).

```cpp
Port p1; // Uninitialized port object.
def foo() -> bool {
    // Implicit port to integer conversion.
    if (p1 == 0)
        return true;
}
```

A.8 Declarations

Declarations, generally speaking, introduce an entity and a name for that entity. When using that name, the declaration is used to determine how to interpret that name.

For example, a variable declaration (see A.8.8) introduces a variable and a name for that variable. When the name for that variable is used in program text, it is used to mean the variable itself.

A definition in Steve is equivalent to a declaration in all but one case. The only case where a definition is distinct from a definition is when the foreign specifier (see A.9.1) is attached to an incomplete function declaration. Declarations have the form:
All declarations introduce a name into a scope. A layout, decoder, table, event, or function declaration must be made in global scope. These are known as global declarations.

Declarations may have a number of specifiers (see A.9) attached to them, given by \( \langle \text{specifier-seq} \rangle \). Specifiers modify the semantics of declarations.

A.8.1 Layout Declaration

Layout declarations are used to define the physical structure of packets in memory. A layout declaration declares a layout type. Layout declarations have the following form:

\[
\langle \text{layout-declaration} \rangle ::= \text{layout} \langle \text{layout-name} \rangle \langle \text{layout-body} \rangle \\
\langle \text{layout-body} \rangle ::= \{ \langle \text{field-declaration} \rangle + \} \\
\langle \text{field-declaration} \rangle ::= \langle \text{field-name} \rangle : \langle \text{type} \rangle ;
\]

136
A field declaration \( F : T \) introduces the name \( F \) into layout scope. \( T \) must be of scalar type (see A.6.1) or layout type (see A.6.8). \(^1\) The type \( T \) specifies the length of the field. Because this is the extent of the usage, there is no reason to support more complex user-defined types here.

Field declarations must occur in the order with which they appear in the an actual instance of a header which the layout represents. Incorrect ordering will result in incorrect extractions by decoders.

Though layout declarations declare layout types, objects of layout type can never be created and functions cannot have parameters or returns of layout type (see A.6.8, 6.2). The following is not legal in Steve.

\[
\text{layout l1 \{} \ldots \}\text{var x : l1; } // \text{Illegal.}
\text{def foo(x : l1) -> l1 \{} \ldots \}\text{ } // \text{Illegal.}
\]

A layout type may only appear as the type of a field declaration in layout scope. This is to allow for handling of nested header structures.

A.8.2 Decoder Declarations

Decoder declarations have the following form:

\[
\langle \text{decoder-declaration} \rangle ::= \text{decoder} \ (\text{decoder-name}) \ [ \text{start} ] \ (\langle \text{layout-id} \rangle) \\
\text{(block-statement)}
\]

The identifier given by \( \langle \text{layout-id} \rangle \) must name a valid layout declaration at global scope. This \( \langle \text{layout-id} \rangle \) is known as the layout rule of the decoder. Dif-

\(^1\)Though \texttt{bool} and \texttt{char} are valid scalar types, they will result in a compiler error if the given field is needed as part of a table's key. This is a limitation that will be adjusted in later revisions.
ferent decoder declarations may use the same layout rule. The layout rule is used to
determine the current view and implicit advances generated by a decoder (see 3.2.4).
Extract declarations (see A.8.3) may only extract fields from this layout.

The optional start keyword shall occur on exactly one decoder declaration
in a given program text. This decoder shall be considered the source (root) of the
pipeline graph during pipeline checking. This decoder shall be the first decoder
applied to a packet context after ingress.

The execution of a decoder is similar to that of a function. Each statement
within ⟨block-statement⟩ is executed in turn.

A.8.3 Extract Declaration

An extract declaration causes a field with the specified field name (see A.2) to be
extracted by the decoder. Extract declarations have the following form:

⟨extract-declaration⟩ ::= extract ⟨field-name⟩;

After this declaration, the name and value of that field may be used in a
⟨field-access-expression⟩ (see A.10.3). An extract declaration must be in de-
coder scope. Attempting to put an extract declaration in any other context shall
result in a compiler error.

The root layout identifier of ⟨field-name⟩ (see A.2) shall be the same layout
as the decoder’s layout rule (see A.8.2). A decoder shall not extract fields from layouts
which are not it’s layout rule. For example, the following is illegal:

// Error: Decoder does not extract ipv4.
decoder eth_d(eth) { extract ipv4.dst; }

138
A.8.4 Rebind Declaration

At certain times, it may be convenient to extract a certain field, but alias that field with a different name than the original. This can be done with the extract-as or rebind declaration. A rebind declaration has the following form:

\[
\langle \text{rebind-declaration} \rangle ::= \text{extract} \langle \text{field-name} \rangle \text{ as } \langle \text{field-name} \rangle;
\]

The rebind declaration has the form extract \( N_1 \) as \( N_2 \) where \( N_1 \) and \( N_2 \) are field names. \( N_1 \) is the original field name and must refer to a field in the layout rule. \( N_2 \) is the alias field name and may refer to any valid field. The types of \( N_1 \) and \( N_2 \) must be the same.

A.8.5 Table and Flow Declarations

Table declarations have the following form:

\[
\langle \text{table-declaration} \rangle ::= \text{table} \langle \text{table-name} \rangle \langle \text{key-declaration-sequence} \rangle [\langle \text{requires-specifier} \rangle ] \langle \text{table-initializer} \rangle
\]

\[
\langle \text{key-declaration} \rangle ::= \langle \text{layout-id} \rangle . \langle \text{field-id} \rangle
\]

\[
| \langle \text{key-declaration} \rangle . \langle \text{field-id} \rangle
\]

\[
| \text{in\_port}
\]

\[
| \text{in\_phys\_port}
\]

\[
\langle \text{requires-specifier} \rangle ::= \text{requires} \langle \text{field-name-sequence} \rangle
\]

\[
\langle \text{table-initializer} \rangle ::= \{[\langle \text{flow-declaration-list} \rangle]\}
\]

\[
\langle \text{flow-declaration-list} \rangle ::= \langle \text{flow-declaration} \rangle
\]

\[
| \langle \text{flow-declaration-list} \rangle \langle \text{flow-declaration} \rangle
\]

Table declarations cause table objects to be created by the runtime system. After loading a Steve generated application, the runtime system (see 7) shall receive a number of table allocation requests from the Steve program corresponding to each
table declaration. These requests are made as part of load-time configuration (see 7.4). Each request shall provide:

- A table name.
- The set of key fields that comprise the table’s key.
- A maximum number of flow entries.

If tables with those properties already exist, the runtime system provides the Steve application with the already existing table. This can happen if multiple Steve applications are run on the same device.

The table’s key fields are a set of one or more fields which together compose a key. A table’s key is given by \(<\text{key-declaration-sequence}>\) in the grammar. Each \(<\text{key-declaration}>\) shall be a valid field name, or be the keywords in_port or in_phys_port. The type of \(<\text{key-declaration}>\) shall be the type of the field, or shall be port type (see A.6.9) in the case of in_port or in_phys_port.

The \(<\text{requires-specifier}>\) is optional. A field name here may also be a \(<\text{key-declaration}>\). This is redundant and does not change semantics. Field names given by \(<\text{key-declaration-sequence}>\) are implicitly required.

Names in \(<\text{key-declaration-sequence}>\) and \(<\text{requires-specifier}>\) are valid inside \(<\text{table-initializer}>\) based on table scope semantics (see A.3.4).

Each table has a set of flow entries declared in Steve using a flow declaration of the following form:
The match fields of a flow entry, that is, values corresponding to a table’s key fields, is given by \( \text{match-field-sequence} \). The type of each \( \text{match-field} \) must be the same type as its corresponding \( \text{key-declaration} \). The number of match fields must be equal to the number of key fields.

All flow entries in the same table must be uniquely identifiable by their match fields and their priority (since Steve only support exact match tables, the priority is the same on all flow entries). If a flow entry being inserted into a table has the same match fields as an already existing flow entry, the previous one is replaced by the one.

Flow entries may optionally have a \( \text{properties-block} \). Properties are additional metadata that can be attached to flow entries. There are currently two properties: \text{timeout} and \text{egress}. The \text{timeout} property specifies the duration (in seconds) that the flow entry will remain the table. The \text{egress} property specifies an egress port that can be accessed in the flow entry. This facilitates the \text{output} \text{egress} action within added flow entries (see A.12.3).
Initial flow entries, that is, the flow entries within \(\text{table-initializer}\), are inserted into tables before any packets are processed. Each initial flow entry is inserted with the order in which they are declared. There may only be one or less \(\text{miss-flow-declaration}\).

A.8.6 Event Declaration

An event declaration has the following form:

\[
\langle \text{event-declaration} \rangle ::= \text{event} \langle \text{event-name} \rangle [ \langle \text{requires-specifier} \rangle ]
\langle \text{block-statement} \rangle
\]

An event declaration defines a special function, known as an event handler, which has access to extractions (see A.8.3) and can use actions (see A.12). They implicitly take a context (see A.5) as a parameter. These are used when an exceptional event has occurred and certain slower actions need to be applied outside of the fast path, e.g. adding and removing flow entries.

An event handler can only be invoked via the raise action. At this time, a copy of the context is passed to the runtime. The runtime shall forward this context and event handler packaged together to the controller port. The controller port waits to receive these contexts and event handlers. It dispatches these contexts to be executed by the event handler in the control plane. Any modifications to the copied context shall not apply to the original. For example, in the following, it is demonstrated that a field modified via \text{set} action only affects a copy and never the original packet.

\[
\text{exact_table \ t1(eth.dst) \{ \{ 0x12_34_56_78_90_ab \} \rightarrow \{ \}
// \text{event1 tries to set eth.dst to 0x0}
\]
raise event1;
// eth.dst will still be 0x12_34_56_78_90_ab
// after the raise action is finished because
// only a copy is passed to event1.
}
}

event event1 requires (eth.dst) {
    // This will not modify the original.
    // Only a copy of the context and packet are
    // operated on.
    set eth.dst = 0x0;
}

Event declarations are subject to meeting the requirements satisfaction property (see 6.1.3) of pipelines, even though they are not pipeline stages.

The execution of an event handler may be immediate or asynchronous. The behavior is runtime implementation specific. The user should have no expectations on the order of execution.

A.8.7 Port Declaration

A port declaration has the following form:

(port-declaration) ::= Port (port-name) [ (port-initializer) ] ;
(port-initializer) ::= = (port-expression)

A port declaration may have an optional (port-initializer) that instantiates it with a port identifier. The (port-initializer) shall be a (port-expr) (see A.10.4). If no initializer is given, the default is a port with identifier 0 (invalid port). Forwarding to the invalid port drops the packet.
A.8.8 Variable Declarations

Variable declarations have the form:

\[
\text{⟨variable-declaration⟩} ::= \text{var} \text{⟨variable-name⟩} : \langle \text{type} \rangle \nn\text{[⟨variable-initializer⟩]};
\]

\[
\text{⟨variable-initializer⟩} ::= = \langle \text{expression} \rangle
\]

A variable declaration may have an optional \text{⟨variable-initializer⟩}. If this initializer is not given, the value of the variable object is default initialized. Default initialization of scalar types shall result in their objects having a binary value of zero.

For a variable declaration \text{var} v1 : T = E1, the type of object v1 is T. The type of E1 must be T or there shall be a type conversion sequence from E1’s type to T.

A.8.9 Function Declarations

A function declaration has the following form:

\[
\text{⟨function-declaration⟩} ::= \text{def} \text{⟨function-name⟩} (\nn\text{[⟨parameter-decl-sequence⟩]} ) \rightarrow \langle \text{type} \rangle \langle \text{block-statement} \rangle
\]

\[
\text{⟨parameter-declaration⟩} ::= \langle \text{parameter-name} \rangle : \langle \text{type} \rangle
\]

A function \text{def} F(P1 : T1, P2 : T2, \ldots, Pn : Tn) \rightarrow T \{ \ldots \} has function type (see A.6.10) (T1, T2, \ldots, Tn) \rightarrow T.

The body of the function, given by \langle \text{block-statement} \rangle, is the set of statements that get executed in order upon calling the function.

There may be multiple declarations of a function with the same name if and only if:
• Each function declaration has a different type.

• Two function declarations of the same name, but different type, may not vary only in their return type.

A.9 Specifiers

Specifiers modify the semantics of a declaration. A sequence of specifiers may optionally appear at the beginning of a declaration. Not all specifiers are allowed on all declarations. Not all specifiers are allowed to appear together before the same declaration. Note that there is currently only one supported specifier. A number of other specifiers will eventually be introduced.

Specifiers have the following form:

⟨specifier⟩ ::= foreign

A.9.1 Foreign Specifier

The foreign specifier can only appear before a ⟨function-declaration⟩. This specifier says that a function has foreign linkage, that is, the function is not defined in this scope but rather in a different program. This function is said to be a foreign function. The name of a foreign function is not mangled.

Foreign functions cannot have a definition, that is, the ⟨function-declaration⟩ cannot have a ⟨block-statement⟩. This is the only case where a declaration varies from a definition.
Foreign linkage is most often used to link against the C runtime library. For example, the following introduces the puts C-function into global scope. Function calls to puts will call the C-function with the same name.

A.10 Expressions

Expressions, generally speaking, perform some type of operation or computation which may return results or have side effects on the state of the program. Steve has the following expressions:

\[ \langle \text{expression} \rangle ::= \langle \text{binary-expression} \rangle \]

A.10.1 Primary Expressions

Primary expressions have no operators. They have the following form:

\[ \langle \text{primary-expression} \rangle ::= \langle \text{identifier-expression} \rangle \\
| \langle \text{field-access-expression} \rangle \\
| \langle \text{literal-expression} \rangle \\
| \langle \text{port-expression} \rangle \]

A.10.2 Identifier Expressions

When an \( \langle \text{identifier} \rangle \) (see A.1) is used where an \( \langle \text{expression} \rangle \) is expected, it is considered an identifier expression. Identifier expressions have the form:

\[ \langle \text{identifier-expression} \rangle ::= \langle \text{identifier} \rangle \]
Identifier expressions refer to a specific declaration of that name. When used as an expression, an identifier returns the entity being referred to. If the identifier refers to an object (see A.4), the type of the object is $T$, and the identifier expression has $T\&$ type.

If an identifier expression is used where an rvalue is expected, then a reference-to-value conversion is applied (see A.7.1).

A.10.3 Field Access Expression

When a field name (see A.2) is used where an $expression$ is expected, it is considered a field access expression. They have the form:

$$<field-access-expression> ::= <field-name>$$

Field access expressions refer to fields extracted by a decoder. They are similar to identifier expressions (see A.10.2), except they specifically refer to extractions.

A field access expression may only occur within the body of a decoder declaration (see A.8.2), a flow declaration (see A.8.5), or an event declaration (see A.8.6). In addition to these limitations, a field access expression may only occur:

- In a decoder declaration if and only if an extract declaration (see A.8.3) extracts a field with the same field name before the field access expression’s use.

- In a flow declaration if and only if the table containing the flow uses the same field name as a key, or lists it in its requires specifier.
In an event declaration if and only if the same field name is in its requires specifier.

If the type of the extraction is T, then the type of a field access expression referring to that extraction shall be T&. When the value of that field is required, then a reference-to-value conversion (see A.7.1) is applied to the field access expression.

A.10.4 Port Expressions

A port expression is either an identifier which refers to a port declaration, or a keyword referring to a number of reserved port names. They have the form:

\[
\begin{align*}
\langle \text{port-expression} \rangle &::= \langle \text{port-id} \rangle \\
&\quad | \langle \text{input-port-expression} \rangle \\
&\quad | \langle \text{input-physical-port-expression} \rangle \\
&\quad | \langle \text{egress-port-expression} \rangle \\
&\quad | \langle \text{all-port-expression} \rangle \\
&\quad | \langle \text{reflow-port-expression} \rangle
\end{align*}
\]

An \(\langle \text{input-port-expression} \rangle\) shall evaluate to the packet’s ingress port.

An \(\langle \text{input-physical-port-expression} \rangle\) shall evaluate to the packet’s physical ingress port.

An \(\langle \text{egress-port-expression} \rangle\) shall evaluate the egress port property set in a flow entry’s properties block (see A.8.5). It shall only occur in the body of a flow entry whose egress property has been set.
The \(<all\text{-}port\text{-}expression>\) shall evaluate to the all port. The all port is a logical port which represents every port on the system.

The \(<reflow\text{-}port\text{-}expression>\) shall evaluate to the reflow port. The reflow port is a logical port which represent re-entry into the pipeline. When output to this port, a packet’s context data is reset and the packet is sent back to the beginning of the pipeline.

A.10.5 Literal Expressions

Literal expressions are expressions which represent constant values. Steve supports decimal integer, binary, hexadecimal, character, and string literals.

A.10.6 Integer Literals

Decimal, binary and hexadecimal literals have the following form:

\[
\begin{align*}
\langle literal-expression \rangle & := \langle decimal-literal \rangle \\
& \quad | \langle binary-literal \rangle \\
& \quad | \langle hexadecimal-literal \rangle \\
\langle decimal-digit \rangle & := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\langle hexadecimal-digit \rangle & := \_ | \langle decimal-digit \rangle | a | b | c | d | e | f \\
\langle binary-digit \rangle & := \_ | 0 | 1 \\
\langle decimal-literal \rangle & := \langle decimal-digit \rangle^+ \\
\langle hexadecimal-literal \rangle & := 0x \langle hexadecimal-digit \rangle^+ \\
\langle binary-literal \rangle & := 0b \langle binary-digit \rangle^+
\end{align*}
\]

Hexadecimal, binary, and decimal integer literals may be used interchangeably to represent the same integer value. The type of an integer literal is by default 32 bit signed integer (see A.6.3). Hexadecimal and binary literals may optionally use
the underscore(_ ) as an organizational separator. This is purely lexical and has no
effect on the value of the literal. For example:

```
10 // Integer literal 10
0b1010 // Binary literal 10
0b10_10 // Binary literal 10 w/ underscore
0x0A // Hexadecimal literal 10
0x00_0A // Hexadecimal literal 10 w/ underscore
```

### A.10.7 Boolean Literals

Boolean literals have boolean type (see A.6.2) and have the values `true` or `false`. They have the following form:

```
⟨boolean-literal⟩ ::= true | false
```

### A.10.8 Character and String Literals

Character and string literals are rarely if ever used in Steve at the moment. Its not
common to need strings in a language directed towards packet processing, which is
largely binary data. However, it may become useful in the future if Steve chooses to
support text-based protocols. Character and string literals have the following form:

```
⟨character-literal⟩ ' ⟨letter⟩ '
⟨string-literal⟩ ::= " ⟨letter⟩+ "
⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
                  V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s |
                  t | u | v | w | x | y | z
```

A character literal has character type (see A.6.4). A string literal has char-
acter block type (see A.6.7). For example:

```
var c1 : char = 'a';
var s1 : char[] = "Hello, world.";
```
A.10.9 Postfix Expressions

Postfix expressions current only include call expressions and array subscript expressions.

\[
\langle \text{postfix-expression} \rangle ::= \langle \text{call-expression} \rangle \\
| \langle \text{subscript-expression} \rangle \\
| \langle \text{primary-expression} \rangle
\]

A.10.10 Call Expression

Call expressions are used to call a function and returns the result of the function. They have the following form:

\[
\langle \text{call-expression} \rangle ::= \langle \text{function-id} \rangle ( [\langle \text{expression-sequence} \rangle] )
\]

The type of a call expression is the return type of the function referred to by \( \langle \text{function-id} \rangle \). The number of expressions in \( \langle \text{expression-sequence} \rangle \) must match the number of parameters required. Their types must also match the types of corresponding parameters, or there must exist a conversion to the corresponding type.

A.10.11 Array Subscript Expression

Array subscript expressions return the value of the element in a given position within an array or block. They have the form:

\[
\langle \text{subscript-expression} \rangle ::= \langle \text{variable-id} \rangle [\langle \text{expression} \rangle]
\]

The \( \langle \text{expression} \rangle \) must have integer type. The type of the object referred to by \( \langle \text{variable-id} \rangle \) must be array or block. If the evaluation of \( \langle \text{expression} \rangle \)
is larger than the extent of the array or block, or it is negative, then behavior is undefined.

A.10.12 Unary Expressions

Unary expressions have a single operator and a single operand. They have the following form:

\[
\langle \text{unary-expression} \rangle ::= \langle \text{not-expression} \rangle \\
| \langle \text{negative-expression} \rangle \\
| \langle \text{postfix-expression} \rangle
\]

A.10.13 Not Expression

The not expression is used to take the logical-not of a boolean value. The not expression has the form:

\[
\langle \text{not-expression} \rangle ::= ! \langle \text{expression} \rangle
\]

Given a not expression !N, both N and the result of !N shall have boolean type. If the value of N is true, the result shall be false. If the value of N is false, the result shall be true.

A.10.14 Negative Expression

The negative expression is used to change the sign of an integer value. Negative expressions have the following form:

\[
\langle \text{negative-expression} \rangle ::= - \langle \text{expression} \rangle
\]

Given an expression -N, N shall have integer type and -N shall have signed integer type. The evaluation shall be equal to N * -1.
A.10.15 Binary Expressions

Binary expressions have a single operator and two operands. They have the following form:

\[
\text{binary-expr} ::= \text{multiplicative-expression} \\
| \text{additive-expression} \\
| \text{bitshift-expression} \\
| \text{ordering-expression} \\
| \text{equality-expression} \\
| \text{bitwise-and-expression} \\
| \text{bitwise-xor-expression} \\
| \text{bitwise-or-expression} \\
| \text{logical-and-expression} \\
| \text{logical-or-expression} \\
| \text{unary-expression}
\]

The precedence of each operator is enforced by the grammar. The operators with the highest precedence (i.e. the lowest number) get evaluated first. Operator precedence is as follows:

1. Unary expressions
2. Multiplicative expressions
3. Additive expressions
4. Bit-shift expressions
5. Ordering expressions
6. Equality expressions
7. Bit-wise And
8. Bit-wise Xor
9. Bit-wise Or
10. Logical And
11. Logical Or
A.10.16 Multiplicative Expression

Multiplicative expressions perform multiplication, division, and modulo operations. They have the following form:

\[
\langle \text{multiplicative-expression} \rangle ::= \langle \text{multiplicative-expression} \rangle \langle \text{multiplicative-operator} \rangle \langle \text{unary-expression} \rangle \\
| \langle \text{unary-expression} \rangle \\
\langle \text{multiplicative-operator} \rangle ::= * | / | \%
\]

Both operands must be of integer type. Arithmetic conversion (see A.7.3) is applied to both operands so that they both have type T, where T is the smallest integer type capable of representing both operands. The result shall have type T.

A.10.17 Additive Expression

Additive expressions perform addition and subtraction operations. They have the following form:

\[
\langle \text{additive-expression} \rangle ::= \langle \text{additive-expression} \rangle \langle \text{add-operator} \rangle \langle \text{multiplicative-expression} \rangle \\
| \langle \text{multiplicative-expression} \rangle \\
\langle \text{add-operator} \rangle ::= + | -
\]

Both operands must be of integer type. Arithmetic conversion (see A.7.3) is applied to both operands so that they both have type T, where T is the smallest integer type capable of representing both operands. The result shall have type T.

A.10.18 Bit-shift Expression

The bit-shift expression shifts the bits of an integer either left or right. They have the form:
Given a bit-shift expression $E_1 << E_2$ or $E_1 >> E_2$, the result is $E_1$ shifted either left or right, respectively. Vacated bit positions are filled with zeros. Bits shifted off either end are discarded. The type of both operands and the result shall be unsigned integer type.

### A.10.19 Ordering and Equality Expression

Ordering expressions check whether or not the value of two operands are: less than, less than or equal, greater than, or greater than or equal. Equality expressions check whether or not the value of two operands are equal or not equal. The result is always true or false. These expressions have the form:

\[
\text{ordering-expression} ::= \text{ordering-expression} \text{ordering-operator} \\
\text{bitshift-expression} \mid \text{bitshift-expression} \\
\text{ordering-operator} ::= < | <= | > | >= \\
\text{equality-expression} ::= \text{equality-expression} \text{equality-operator} \\
\text{ordering-expression} \mid \text{ordering-expression} \\
\text{equality-operator} ::= == | != 
\]

Given a type $T$, assuming the type of the first operand is $T_1$, and the type of the second is $T_2$, there must exist a sequence of conversions from $T_1$ to $T$ and from $T_2$ to $T$. In other words, the types of both operands must be convertible to some common type. The type of the result shall be boolean type.
A.10.20 Bitwise Expressions

The bitwise expressions perform bitwise AND, XOR, or OR operations. They have the form:

\[
\begin{align*}
\text{bitwise-and-expression} & := \text{bitwise-and-expression} \& \text{equality-expression} \\
& \quad | \text{equality-expression} \\
\text{bitwise-xor-expression} & := \text{bitwise-xor-expression} \mid \text{bitwise-and-expression} \\
& \quad | \text{bitwise-and-expression} \\
\text{bitwise-or-expression} & := \text{bitwise-or-expression} \^ \text{bitwise-xor-expression} \\
& \quad | \text{bitwise-xor-expression}
\end{align*}
\]

Both operands must be of integer type. Arithmetic conversion (see A.7.3) is applied to both operands so that they both have type \(T\), where \(T\) is the smallest integer type capable of representing both operands. The type of the result shall be \(T\).

A.10.21 Logical And and Logical Or Expression

The type of both operands in logical-and and logical-or expressions shall be boolean type. The result shall have boolean type. They have the following form:

\[
\begin{align*}
\text{logical-and-expression} & := \text{logical-and-expression} \&\& \text{bitwise-or-expression} \\
& \quad | \text{bitwise-or-expression} \\
\text{logical-or-expression} & := \text{logical-or-expression} \mid\mid \text{logical-and-expression} \\
& \quad | \text{logical-and-expression}
\end{align*}
\]
A.11 Statements

Statements are pieces of code which are executed in sequence, and have some side-effects, but unlike expressions do not evaluate to any value or return any results. They have the form:

\[
\langle \text{statement} \rangle ::= \langle \text{block-statement} \rangle \\
| \langle \text{expression-statement} \rangle \\
| \langle \text{declaration-statement} \rangle \\
| \langle \text{assign-statement} \rangle \\
| \langle \text{if-then-statement} \rangle \\
| \langle \text{if-else-statement} \rangle \\
| \langle \text{match-statement} \rangle \\
| \langle \text{case-statement} \rangle \\
| \langle \text{while-statement} \rangle \\
| \langle \text{break-statement} \rangle \\
| \langle \text{continue-statement} \rangle \\
| \langle \text{return-statement} \rangle \\
| \langle \text{action} \rangle
\]

A.11.1 Block Statement

A block statement is a compound statement which contains a sequence of statements that are executed in order. A block statement introduces block scope (see A.3.2). Block statements may be nested within each other. A block statement may contain no statements It has the form:

\[
\langle \text{block-statement} \rangle ::= \{ \langle \text{statement} \rangle \}
\]

A.11.2 Expression Statement

An expression statement executes an expression. It has the form:

\[
\langle \text{expression-statement} \rangle ::= \langle \text{expression} \rangle;
\]
The \textit{expression} is executed and any return value it might have is discarded. It may have had side effects on the program.

A.11.3 Declaration Statement

A declaration statement introduces a declaration at compile time. Declaration statements have the following form:

\[
\textit{declaration-statement} ::= \textit{declaration};
\]

Because a statement generally only appears in a block statement, global declarations (see A.8) may not appear in declaration statements.

A.11.4 Assignment Statements

Assignment statements allow the contents of one object to be replaced with the contents of another. Assignment statements have the form:

\[
\textit{assign-statement} ::= \textit{identifier-expression} = \textit{expression};
\]

The \textit{identifier-expression} must refer to a valid object declaration. That object must be assignable. Only variables and ports are assignable. When an assignment statement is executed, the result \textit{expression} shall be copied into that object.

Given an assignment statement \(E_1 = E_2\), \(E_1\) must have type \(T\&\). \(E_2\) must have type \(T\) or there must exist a conversion sequence to \(T\).
A.11.5 If and If-else

If and if-else statements have the same semantics as almost all other languages. They have the form:

\[
\begin{align*}
\text{if-statement} & \ ::= \ if \ (expression) \ statement \\
\text{if-else-statement} & \ ::= \ if \ (expression) \ statement \ else \ statement
\end{align*}
\]

Given an if statement of the form \( \text{if} \ (E1) \ S1 \), the type of \( E1 \) must be boolean. \( S1 \) will only execute if the value of \( E1 \) is \text{true}.

For an if-else statement of the form \( \text{if} \ (E1) \ S1 \ else \ S2 \), the type of \( E1 \) must be boolean. \( S1 \) will only execute if the value of \( E1 \) is \text{true}, otherwise \( S2 \) executes.

A.11.6 Match

Match statements are similar to C-like switch statements with some minor differences. They have the form:

\[
\begin{align*}
\text{match-statement} & \ ::= \ match \ (expression) \ match-body \\
\text{match-body} & \ ::= \ \{ \ case-statement \}+ \\
\text{case-statement} & \ ::= \ case \ (case-label) : \ case-body \\
\text{case-label} & \ ::= \ case \ literal-expression : \\
& | \ miss : \\
\text{case-body} & \ ::= \ statement
\end{align*}
\]

Match statements do not have fall-through behavior like switch statements. The condition \( expression \) is evaluated; if the value is equal to the value of a \( case-label \), then the corresponding \( case-body \) is executed. The match statement then completes.
The type of \( \textit{expression} \) in \( \textit{match-statement} \) shall be integer type \( T \).

The type of \( \textit{literal-expression} \) in all \( \textit{case-statements} \) shall also be \( T \).

A.11.7 While Loops

While loops work in the same way as all C-like languages. They have the following form:

\[
\textit{while-statement} ::= \textbf{while} (\langle \textit{expression} \rangle) \langle \textit{statement} \rangle
\]

The type of \( \textit{expression} \) shall be boolean. The value of \( \textit{expression} \) is evaluated at the beginning of every loop. While it evaluates to \textit{true}, \( \textit{statement} \) shall be executed.

A.11.8 Break and Continue

Break and continue may only appear in certain contexts. As of now, they may only appear within a while statement.

A break statement can be used to exit a loop prematurely. When a break statement is executed, the program branches to and executes the statement immediately following the while statement.

A continue statement can be used to branch to the first statement in the while loop if \( \textit{statement} \) is a block statement. These have the form:

\[
\textit{break-statement} ::= \textbf{break} ;
\]
\[
\textit{continue-statement} ::= \textbf{continue} ;
\]
A.11.9 Return Statement

A return statement returns a value from a function and terminates the execution of that function. It has the form:

\[
\begin{align*}
\text{(return-statement)} & ::= \text{return (expression)}; \\
\end{align*}
\]

Return statements may only appear within a function body. Executing a return statement shall evaluate \((\text{expression})\) and a copy of its value is returned to the caller. The type of \((\text{expression})\) shall be the same as the function’s return type, or there shall exist a conversion sequence to the return type.

A.12 Actions

An action is a special statement that may affect the state of a packet, its context, or a table. Because actions are specific to packet handling, an action may only appear in the body of a decoder, event, or flow entry. Actions have the form:

\[
\begin{align*}
\text{(action-statement)} & ::= \text{(decode-action)} \\
& \quad | \text{(goto-action)} \\
& \quad | \text{(output-action)} \\
& \quad | \text{(flood-action)} \\
& \quad | \text{(drop-action)} \\
& \quad | \text{(clear-action)} \\
& \quad | \text{(set-field-action)} \\
& \quad | \text{(insert-flow-action)} \\
& \quad | \text{(remove-flow-action)} \\
& \quad | \text{(raise-action)} \\
& \quad | \text{(write-action)}
\end{align*}
\]
A.12.1 Decode Action

The decode action is a stage transitioning action. It transitions the context (see A.5) from the current stage (which may be a decoder, table, or event) to a new decoding stage. It has the form:

\[
\langle \text{decode-action} \rangle ::= \text{decode} \langle \text{decoder-id} \rangle [\langle \text{advance-specifier} \rangle] ;
\]

\[
\langle \text{advance-specifier} \rangle ::= \text{advance} \langle \text{expression} \rangle
\]

A decode action transfers control of the context it is operating on to the decoder named by \( \langle \text{decoder-id} \rangle \). The semantics of transferring control are similar to calling a function and passing the context as a parameter. If a decode action appears within the body of a decoder (see A.8.2), then a view advance (see 3.2.4) is applied before the decode action transfers control to the next stage.

A decode action may optionally have an \( \langle \text{advance-specifier} \rangle \), if and only if it occurs within a decoder. Given a decode action of the form \text{decode d1 advance E1}, the type of \( \text{E1} \) shall be unsigned integer. The result of \( \text{E1} \) (in bytes) shall be added to the view index. The reason the shift is by bytes and not by bits is because generally speaking, headers are byte aligned.

A.12.2 Goto Action

The goto action is a stage transition action, similar to the decode action. A goto action transitions the context (see A.5) from the current stage (which may be a decoder, table, or event) to a new table matching stage.

\[
\langle \text{goto-action} \rangle ::= \text{goto} \langle \text{table-id} \rangle [\langle \text{advance-specifier} \rangle] ;
\]
A goto action transfers control of the context it is operating on to the table matching stage referred to by \( \text{table-id} \). The context is dispatched to the runtime (see 7) which composes the query key from extracted fields. The query key is then matched against flow entries. All matching flow entries are collected. The one with the highest priority is executed. Because Steve currently only supports exact match tables, there will only ever be one match. The flow entry’s actions are then executed. This body may include another stage transitioning action.

If a goto action appears within the body of a decoder (see A.8.2), then a view advance (see 3.2.4) is applied before transferring to the table matching stage.

A goto action may optionally have an \( \text{advance-specifier} \), if and only if it occurs with the body of a decoder. Given a goto action of the form \texttt{goto d1 advance E1}, the type of \( \text{E1} \) shall be unsigned integer. The result of \( \text{E1} \) (in bytes) shall be added to the view index.

A.12.3 Output Action

An output action forwards a copy of the packet to the designated port. It has the form:

\[
\langle \text{output-action} \rangle ::= \text{output } \langle \text{port-expression} \rangle ;
\]

When an output action is executed, control of the packet is returned to the runtime (see 7) which creates a copy of the packet. The copy is forwarded by the runtime through the port given by \( \langle \text{port-expression} \rangle \).
If destination is the all port, copies of the packet are sent on all known ports on the system. If destination is the flood port, copies of the packet are sent on all ports except the packet’s ingress port. If the destination is the reflow port, then the runtime will send the packet back to the beginning of the pipeline. If the destination is a specific port, the packet is forwarded on that port. After this completes, pipeline processing continues on the original packet.

The output semantics here are a little unintuitive. Some may have the expectation that outputting a packet terminates pipeline processing and immediately forwards the original. This is not the case. Pipeline processing only terminates on the original packet if a stage does not send it to another stage, and does not apply any more actions to it.

At this point egress processing (see 3.2.12) would begin. Any actions written to the action list would be executed. Written output actions modify the egress port field in the context. The original packet is forwarded to the port saved in this field.

A.12.4 Drop Action

The drop action is used to delete the packet and immediately terminate pipeline processing. The drop action has the following form:

\[
\text{drop-action} ::= \text{drop} ;
\]

The drop action is implicitly applied to any packet who has reached egress processing (see 3.2.12), but which does not have an output action (see A.12.3) written to its action list.
A.12.5 Clear Action

A clear action removes all actions from the context’s action list. If the action list is empty, nothing is done. The clear action has the following form:

\[ \text{clear-action} ::= \text{clear} ; \]

A.12.6 Set Action

A set action copies a given value into a given field. The set action has the following form:

\[ \text{set-action} ::= \text{set} \left( \text{field-access-expression} \right) = \left( \text{expression} \right) ; \]

If \( \text{field-access-expression} \) has type \( T\& \), then the result of evaluating \( \text{expression} \) shall have type \( T \), or there shall exist a conversion sequence from its type to \( T \). This prevents the length of the value from exceeding the length of the field.

A.12.7 Insert Action

Insert action will add a new flow entry into a table. They may also be used to update an existing flow entry into a table. They have the form:

\[ \text{insert-action} ::= \text{insert into} \left( \text{table-id} \right) \left( \text{flow-declaration} \right) ; \]

The flow entry represented by the \( \text{flow-declaration} \) is inserted into the table referred to by \( \text{table-id} \). Each \( \text{match-field} \) in \( \text{match-field-sequence} \) is evaluated. The number and type of each \( \text{match-field} \) must match the number and type of each \( \text{key-field} \) in the table. Type conversions may be applied to the types to make them match.
If a flow entry already in that table has the same match fields as the flow entry being inserted, then the prior one is replaced with the new one. All properties of the old flow entry are replaced with the properties of the inserted flow entry. If no properties are given in the inserted flow entry, the properties are the default values (no timeout and no egress port).

A miss case flow entry may be given as \textit{flow-declaration}. The same insertion semantics apply here as well.

A.12.8 Remove Action

A remove action will remove an existing flow entry from a table. They have the form:

\[
\textit{remove-action} ::= \textit{remove from }\langle\textit{table-id}\rangle\langle\textit{match-field-sequence}\rangle; \\
| \langle\textit{remove-miss-action}\rangle
\]

Each \textit{match-field} in \textit{match-field-sequence} is evaluated. The table referred to by \textit{table-id} is searched for all flow entries with the same match fields. All flow entries found are removed from the table. If none are found, nothing is done.

Miss case flow entries may also be removed from a table. Miss case removal has the form:

\[
\textit{remove-miss-action} ::= \textit{remove from }\langle\textit{table-id}\rangle\textit{miss};
\]

When a miss case is removed, it is replaced by the default miss case. The default miss case has no timeout and no egress port. Its body has exactly one action: the drop action.
A.12.9 Write Action

A write action writes another action to a context’s action list. They have the form:

\[ \text{write-action} ::= \text{write} \langle \text{action-statement} \rangle ; \]

All actions written to the context’s are executed upon egress processing (see 3.2.12) in the order with which they were written. Written actions may have different semantics when executed during egress processing. Not all actions can be written. The actions which are writable may expand in later versions. The following are writable actions and their semantics during egress processing.

- Set. The set action works same as it would if executed immediately (see A.12.9).

- Output. The output action lists the egress port field in the context. It does not immediately forward the packet upon execution. When every action in the action list is completed, the original packet is forwarded (not a copy).

A.12.10 Raise Action

A raise action is used to raise an event which causes the context to be dispatched to the controller port. The packet is then sent to an event handler with the same name. The event handler gets executed by the control plane. The raise action has the form:

\[ \text{raise-action} ::= \text{raise} \langle \text{event-id} \rangle [(\text{advance-specifier})] ; \]

When executed, the runtime (see 7) will create a message containing the context and the event handler function. This message is sent to the controller port. The controller port is connected to a controller. That controller shall execute the
event handler, passing in the context as the argument. This action may be taken synchronously or asynchronously. The choice is runtime implementation specific.

A.13 Steve Programs

This section contains all Steve applications written throughout this thesis in their entirety.

A.13.1 Wire

```cpp
layout ethernet {
    dst : uint (48);
    src : uint (48);
    type : uint (16);
}

Port p1;
Port p2;

deriver start eth_d(ethernet) {
    // Port variables, p1 and p2, "remember" ports.

    // Whenever a packet is handled, check if p1 and p2 are set.
    // If neither are, set p1 equal to the ingress port.
    if (p1 == 0 && p2 == 0)
        p1 = in_port;
    // If p1 is set, and p2 isn’t, set p2 to the ingress port.
    else if (p1 != 0 && p2 == 0)
        p2 = in_port;

    // Now we decide which packet to forward to.
    // If the ingress port is p1, and p2 is set, forward to p2.
    if (in_port == p1 && p2 != 0)
        output p2;
    // If the ingress port is p2, and p1 is set, forward to p1.
    if (in_port == p2 && p1 != 0)
```
output p1;
// If both are not set yet, do nothing and implicitly drop.
}

A.13.2 MAC Bridge

////////////////////////////////////////////////////////////////////////////////
// MAC Learning Switch
////////////////////////////////////////////////////////////////////////////////

layout ethernet {
  dst : uint(48);
  src : uint(48);
  type : uint(16);
}

decoder start eth_d(ethernet) {
  extract ethernet.dst;
  extract ethernet.src;
  goto learn;
}

// This event will do the "learning" through flow inserts.
event learn_mac
  requires(ethernet.src) // It requires the src MAC
{
  // First we insert the src of the packet
  // into the learn table so we don’t keep
  // trying to learn something we already have.
  insert into learn
  { ethernet.src } -> [timeout = 60] { goto forward; }
  // Next we insert the src of the current packet
  // into the forward table.
  //
  // The forward table matches on the dst field of a packet.
  // What we are doing is saying any packet whose dst is equal
  // to this packet’s src is forwarded to this packet’s
  // ingress port.
  insert into forward
  { ethernet.src } ->
  [timeout = 60, egress = in_port]
{
  // We set the egress property to the current
  // packet’s in_port. Future packets will be forwarded to
  // the current packet’s ingress port.
}
output egress;
};

exact_table learn(ethernet.src) {
  miss ->
  {
    raise learn_mac;
    goto forward;
  }
}

exact_table forward(ethernet.dst) {
  miss ->
  {
    output flood;
  }
}

A.13.3 IPv4 Routing

////////////////////////////////////////////////////////////////////////
// IPV4 Learning Switch
////////////////////////////////////////////////////////////////////////

layout ethernet {
  src : uint(48);
  dst : uint(48);
  type : uint(16);
}

layout ipv4 {
  version_ihl : uint(8);
  dscp_ecn : uint(8);
  len : uint(16);
  id : uint(16);
  fragment : uint(16);
  ttl : uint(8);
  protocol : uint(8);
  checksum : uint(16);
  src : uint(32);
  dst : uint(32);
}

decoder start eth_d(ethernet) {


extract ethernet.type;
if (ethernet.type >= 0x600)
    // The next header is IPv4 if type field is 0x800.
    match (ethernet.type) {
        case 0x800: decode ipv4_d;
    }
    // If it's not IPv4, processing ends and the packet is
    // implicitly dropped.
}

decoder ipv4_d(ipv4) {
    // We actually need all fields to confirm the checksum.
    extract ipv4.version_ihl;
    extract ipv4.dscp_ecn;
    extract ipv4.len;
    extract ipv4.id;
    extract ipv4.fragment;
    extract ipv4.ttl; // Use time-to-live to decrement
    extract ipv4.protocol;
    extract ipv4.checksum;
    extract ipv4.src;
    extract ipv4.dst;

    // Calculate a checksum.
    var checksum : uint(16) =
        ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, 
        ipv4.len, ipv4.id, ipv4.fragment, ipv4.ttl, 
        ipv4.protocol, ipv4.src, ipv4.dst);
    // Check the checksum against the header's checksum.
    if (checksum != ipv4.checksum)
        drop;
    // Drop time-to-live expired packets.
    if (ipv4.ttl == 0)
        drop;
    set ipv4.ttl = ipv4.ttl - 1; // Decrement ttl.
    // We've changed the ttl, we must set a new checksum.
    set ipv4.checksum =
        ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, 
        ipv4.len, ipv4.id, ipv4.fragment, 
        ipv4.ttl, ipv4.protocol, ipv4.src, ipv4.dst);
    // Proceed to the learn table after advancing by ihl
    goto learn advance (ipv4.version_ihl & 0x0f) * 4;
}

// This event handles the actual "learning."
event learn_ip
requires(ipv4.src) // It will learn src IP addresses.
{
  // This first entry prevents the same address from causing
  // this event twice. Sends the packet straight to routing.
  insert into learn
  { ipv4.src } -> [timeout = 30] { goto route; }
  // This establishes the IP address to port mapping.
  // Any packet whose dst address matches the current packet's
  // src address will be forwarded to the current packet's
  // ingress port.
  insert into route
  { ipv4.src } ->
  [timeout = 30, egress = in_port]
  {
    output egress;
  }
}

exact_table learn(ipv4.src) {
  miss -> {
    raise learn_ip;
    goto route;
  }
}

// This ultimately decides where to forward packets based on
// their destination IP.
exact_table route(ipv4.dst) {
  miss -> {
    output flood; // Flood on all unlearned addresses.
  }
}

def ipv4_checksum(vihl : uint(8), dscp_ecn : uint(8),
  len : uint(16), id : uint(16), frag : uint(16),
  ttl : uint(8), proto : uint(8), src : uint(32),
  dst : uint(32) ) -> uint(16)
{
  // Merge fields into 16-bit values.
  var merge1 : uint(16) = vihl << 8;
  merge1 = merge1 | dscp_ecn;
  var merge2 : uint(16) = ttl << 8;
  merge2 = merge2 | proto;
  // Split fields into 16-bit values.
  var split_src1 : uint(16) = src >> 16;
  // Get the lower 16 bits.
var split_src2 : uint(16) = src & 0x0000_ffff;
// Get the upper 16 bits.
var split_dst1 : uint(16) = dst >> 16;
// Get the lower 16 bits.
var split_dst2 : uint(16) = dst & 0x0000_ffff;
// Accumulated sum.
var acc : uint(32) = 0;
acc = acc + merge1 + len + id + frag + merge2 +
    split_src1 + split_src2 + split_dst1 + split_dst2;
// Perform the 1’s complement sum wraparound.
var acc1 : uint(16) = acc >> 16;
var acc2 : uint(16) = acc & 0x0000_ffff;
acc2 = acc1 + acc2;
acc2 = acc2 ^ 0xffff_ffff;
return acc2;
}

A.13.4 Stateless Firewall

/////////////////////////////////////////////////////////////////////
// Firewall
/////////////////////////////////////////////////////////////////////

layout eth {
    dst : uint(48);
    src : uint(48);
    type : uint(16);
}

layout ipv4 {
    version_ihl : uint(8);
    dscp_ecn : uint(8);
    len : uint(16);
    id : uint(16);
    fragment : uint(16);
    ttl : uint(8);
    protocol : uint(8);
    checksum : uint(16);
    src : uint(32);
    dst : uint(32);
    // Ignore options.
}

layout udp {
    src : uint(16);
}
dst : uint(16);
len : uint(16);
checksum : uint(16);
}

layout tcp {
src : uint(16);
dst : uint(16);
ack : uint(32);
seq : uint(32);
control : uint(16);
window : uint(16);
checksum : uint(16);
urgent_ptr : uint(16);
// Ignore options.
}

decoder start eth_d(eth) {
extract eth.type;
// Check for IPv4. Ignore IPv6 for now.
match(eth.type) {
  case 0x800: decode ipv4_d;
}
// Don’t worry about other kinds of packets.
}

decoder ipv4_d(ipv4) {
// We actually need all fields to confirm the checksum.
extract ipv4.version_ihl; // Use this to get header length.
extract ipv4.dscp_ecn;
extract ipv4.len;
extract ipv4.id;
extract ipv4.fragment;
extract ipv4.ttl; // Use time-to-live to decrement
extract ipv4.protocol;
extract ipv4.checksum;
extract ipv4.src;
extract ipv4.dst;

// Calculate a checksum.
var checksum : uint(16) =
  ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, ipv4.len,
                ipv4.id, ipv4.fragment, ipv4.ttl, ipv4.protocol,
                ipv4.src, ipv4.dst);
// Check the checksum against the header’s checksum.
if (checksum != ipv4.checksum)
// Drop time-to-live expired packets.
if (ipv4.ttl == 0)
    drop;

set ipv4.ttl = ipv4.ttl - 1; // Decrement ttl.
// We’ve changed the ttl, we must set a new checksum.
set ipv4.checksum =
    ipv4_checksum(ipv4.version_ihl, ipv4.dscp_ecn, ipv4.len, ipv4.id, ipv4.fragment,
                  ipv4.ttl, ipv4.protocol, ipv4.src, ipv4.dst);

var hdr_len : uint(8) = (ipv4.version_ihl & 0x0f) * 4;
// Only care about udp and tcp requests,
match (ipv4.protocol) {
    case 0x06: decode tcp_d advance hdr_len;
    case 0x11: decode udp_d advance hdr_len;
}

decoder udp_d(udp) {
    extract udp.dst;
    goto udp_filter;
}

decoder tcp_d(tcp) {
    extract tcp.dst;
    goto tcp_filter;
}

// Only route if the port# is 80 or 443
// Log the ip src address of all blocked and all allowed.
exact_table udp_filter(udp.dst) requires(ipv4.src) {
    { 80 } -> { goto learn; }
    { 443 } -> { goto learn; }
    miss -> { drop; }
}

// Only route if the port# is 80 or 443
// Log the IP address of all blocked and all allowed.
exact_table tcp_filter(tcp.dst) requires(ipv4.src) {
    { 80 } -> { goto learn; }
    { 443 } -> { goto learn; }
    miss -> { drop; }
}

// This event handles the actual "learning."
event learn_ip
requires(ipv4.src) // It will learn src IP addresses.
{
  // This first entry prevents the same address from causing
  // this event twice. Sends the packet straight to routing.
  insert into learn
  { ipv4.src } -> [timeout = 30] { goto route; };
  // This establishes the IP address to port mapping.
  // Any packet whose dst address matches the current packet’s
  // src address will be forwarded to the current packet’s
  // ingress port.
  insert into route
  { ipv4.src } ->
  [timeout = 30, egress = in_port]
  {
    output egress;
  };
}

// Learning table.
exact_table learn(ipv4.src) {
  miss -> {
    raise learn_ip;
    goto route;
  }
}

// This ultimately decides where to forward packets based on
// their destination IP.
exact_table route(ipv4.dst)
  requires(ipv4.ttl)
{
  miss -> {
    output flood; // Flood on all unlearned addresses.
  }
}

def ipv4_checksum(vihl : uint(8), dscp_ecn : uint(8),
  len : uint(16), id : uint(16), frag : uint(16),
  ttl : uint(8), proto : uint(8), src : uint(32),
  dst : uint(32) ) -> uint(16)
{
  // Merge fields into 16-bit values.
  var merge1 : uint(16) = vihl << 8;
  merge1 = merge1 | dscp_ecn;
  var merge2 : uint(16) = ttl << 8;
  merge2 = merge2 | proto;

// Split fields into 16-bit values.
var split_src1 : uint(16) = src >> 16;
// Get the lower 16 bits.
var split_src2 : uint(16) = src & 0x0000_ffff;
// Get the upper 16 bits.
var split_dst1 : uint(16) = dst >> 16;
// Get the lower 16 bits.
var split_dst2 : uint(16) = dst & 0x0000_ffff;
// Accumulated sum.
var acc : uint(32) = 0;
acc = acc + merge1 + len + id + frag + merge2 +
    split_src1 + split_src2 + split_dst1 + split_dst2;
// Perform the 1's complement sum wraparound.
var acc1 : uint(16) = acc >> 16;
var acc2 : uint(16) = acc & 0x0000_ffff;
acc2 = acc1 + acc2;
acc2 = acc2 ^ 0xffff_ffff;
return acc2;