Спецификация языка P4$_{16}$, версия 1.2.2

The P4$_{16}$ Language Specification
version 1.2.2
The P4 Language Consortium
2021-05-17

Аннотация

P4 - язык программирования для уровня данных сетевых устройств. В этом документе приведено точное определение языка P4$_{16}$, который является результатом пересмотра в 2016 г. языка P4 (http://p4.org). Документ предназначен для разработчиков, создающих компиляторы, имитаторы, среды разработки (IDE) и отладчики для программ P4. Документ может также быть интересен программистам P4, желающим более глубоко понять синтаксис и семантику языка.

Оглавление

1. Сфера применения

Эта спецификация задаёт структуру и интерпретацию программ на языке $P4_{16}$. Определения включают синтаксис, семантику и требования к соответствию для реализаций.

Документ не задаёт:

- механизмы компиляции, загрузки и исполнения программ P4 в системах обработки пакетов;
- механизмы передачи данных из одной системы обработки пакетов в другую;
- механизмы, с помощью которых уровень управления поддерживает таблицы «совпадение-действие» и другие зависящие от состояния объекты, определённые программами P4;
- размер и сложность программ P4;
- минимальные требования к системам обработки пакетов, соответствующим спецификации.

Понятно, что некоторые реализации не смогут полностью поддерживать описанное здесь поведение во всех ситуациях или могут предоставлять варианты повышения производительности или обработки более крупных программ за счёт снижения гарантий безопасности. Таким реализациям следует документировать свои отличия от этой спецификации.

2. Термины, определения, символы

Ниже приведены определения используемых в документе терминов.

Architecture - архитектура
Набор программируемых на P4 компонентов и интерфейсов уровня данных между ними.

Control plane - уровень (плоскость) управления
Класс алгоритмов и соответствующих входных и выходных данных, связанных с обеспечением и настройкой конфигурации плоскости данных.

Data plane - уровень (плоскость) данных
Класс алгоритмов, описывающих преобразования пакетов в системах пакетной обработки.

Metadata - метаданные
Промежуточные данные, создаваемые в процессе выполнения программы P4.

Packet - пакет
Форматированный блок данных, передаваемый в сети с коммутацией пакетов.

Packet header - заголовок пакета
Форматированные данные в начале пакета. Пакет может содержать последовательность заголовков, представляющих различные сетевые протоколы.

Packet payload - данные пакета
Данные, следующие после заголовков пакета.

Packet-processing system - система обработки пакетов
Система обработки данных, предназначенная для работы с сетевыми пакетами. В общем случае система обработки пакетов реализует алгоритмы плоскостей данных и управления.

Target - целевая платформа (цель)
Система обработки пакетов, способная выполнять программы P4.

Термины, определённые в этом документе, не следует трактовать как косвенно связанные с ними термины других документов. И наоборот, не определённые здесь термины следует интерпретировать в соответствии с общепризнанными источниками, такими как IETF RFC.

3. Обзор

P4 представляет собой язык, описывающий обработку пакетов в плоскости данных программируемого элемента пересылки, такого как программный или аппаратный коммутатор, сетевой адаптер, маршрутизатор или специализированная сетевая платформа. Имя P4 возникло из названия статьи, где язык был предложен - «Programming Protocol-independent Packet Processors»[1]. Хотя P4 изначально был предназначен для программируемых коммутаторов, область его применения расширилась и сейчас охватывает широкий спектр устройств. В оставшейся части документа такие устройствами будут называться целевыми платформами или целями (target).

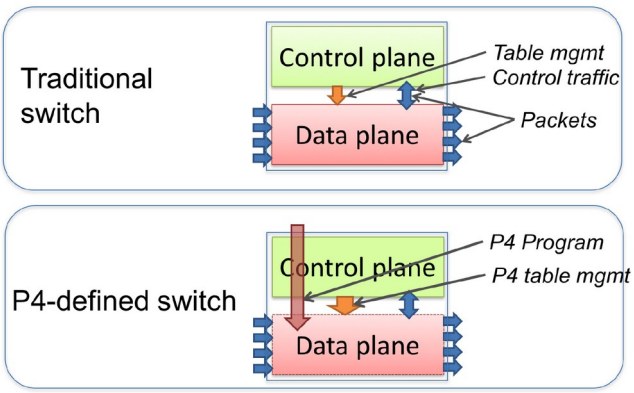

Рисунок 1. Традиционные и программируемые коммутаторы.

Многие платформы реализуют плоскости данных и управления. Язык P4 предназначен лишь для задания функциональности в плоскости данных. Программы P4 также частично определяют интерфейс взаимодействия между плоскостями данных и управления, но P4 нельзя использовать для описания функциональности плоскости управления

[1] *Перевод статьи доступен на сайте www.protokols.ru. Прим. перев.*

целевой платформой. В оставшейся части документа P4 рассматривается как язык программирования платформы в понимании «программирования плоскости данных».

В качестве примера целевой платформы на рисунке 1 показано различие между традиционным коммутатором с фиксированной функциональностью и программируемым коммутатором P4. В традиционном коммутаторе функциональность плоскости данных задаёт производитель. Плоскость управления контролирует плоскость данных через записи в таблицах (например, таблица маршрутизации), настройку специализированных объектов (например, измерителей) и обработку управляющих пакетов (например, пакеты протоколов маршрутизации) или по асинхронным событиям, таким как смена состояния или уведомления системы обучения.

Программируемый коммутатор P4 отличается от традиционного в двух важных аспектах.

- Функциональность плоскости данных не фиксируется заранее, а задаётся программой P4. Плоскость данных настраивается при инициализации и реализует функциональность, описанную программой (длинная красная стрелка на рисунке), и не имеет встроенных знаний об имеющихся сетевых протоколах.
- Плоскость управления взаимодействует с уровнем данных по тем же каналам, что и в фиксированном устройстве, но набор таблиц и других объектов плоскости данных больше не является фиксированным, поскольку его определяет программа P4. Компилятор P4 генерирует API, используемый плоскостью управления для взаимодействия с плоскостью данных.

Следовательно P4 можно считать независимым от протоколов языком, позволяющим программисту выразить широкий спектр протоколов и других деталей поведения плоскости данных. Базовые абстракции P4 приведены ниже.

- ***Типы заголовков***, описывающие формат (набор и размеры полей) каждого заголовка в пакете.
- ***Синтаксические анализаторы*** (parser) описывают разрешённые последовательности заголовков в принимаемых пакетах, идентификацию таких последовательностей, а также заголовки и поля для извлечения из пакетов.
- ***Таблицы*** связывают заданные пользователем ключи с действиями. Таблицы P4 являются обобщением таблиц традиционных коммутаторов и могут служить для реализации таблиц маршрутизации, таблиц поиска потоков, списков контроля доступа и других, заданных пользователем, типов таблиц, включая сложные решения на основе множества переменных.
- ***Действия*** - фрагменты кода, описывающие манипуляции с полями пакета и метаданными. Действия могут включать данные, представленные плоскостью управления в процессе работы.
- ***Блоки «сопоставление-действие»*** (СД), выполняющие ряд операций:
  - создание ключей поиска из полей пакета или рассчитанных метаданных;
  - поиск в таблицах по созданным ключам и выбор выполняемых действий (включая связанные данные);
  - выполнение выбранного действия.
- ***Поток управления*** выражает императивную программу, описывающую обработку пакетов на целевой платформе, включая зависимые от данных последовательности вызовов блоков СД. Повторная сборка пакетов (deparsing) также может выполняться с использованием потока управления.
- ***Внешние объекты*** - зависимые от архитектуры конструкции, которыми могут манипулировать программы P4 через чётко определённые API, но внутреннее поведение которых задано жёстко (например, блоки контрольных сумм) и не программируется в P4.
- ***Пользовательские метаданные*** - заданные пользователем структуры данных, связанные с пакетом.
- ***Внутренние метаданные***, обеспечиваемые архитектурой, связанной с каждым пакетом (например, входной порт, принявший пакет).

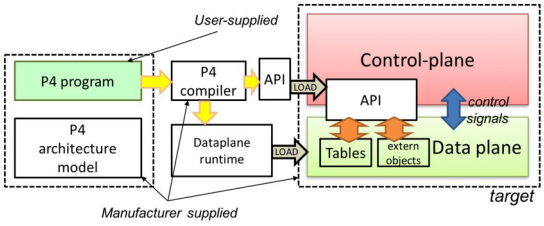

Рисунок 2. Программирование с использованием P4.

На рисунке 2 показан типовой процесс программирования целевой платформы с использованием P4.

Производители платформ обеспечивают аппаратную или программную реализацию модели, определение архитектуры и компилятор P4 для платформы. Программисты P4 создают программы для определённой архитектуры, которая задаёт набор программируемых на P4 компонентов, а также их внешние интерфейсы плоскости данных.

Компиляция набора программ P4 создаёт:

- конфигурацию плоскости данных, реализующую логику пересылки, описанную в программе;
- API для управления состояниями объектов уровня данных из плоскости управления.

P4 является предметно-ориентированным языком, предназначенным для реализации на разных платформах, включая программируемые интерфейсные платы, FPGA, программные коммутаторы и микросхемы ASIC. Поэтому язык ограничен конструкциями, которые могут быть эффективно реализованы на всех таких платформах.

В предположении фиксированной стоимости операций поиска в таблицах и взаимодействия с внешними объектами, все программы P4 (т. е. анализаторы и элементы управления) выполняют постоянное число операций для каждого

принятого и проанализированного байта. Хотя анализаторы могут включать циклы с извлечением некого заголовка в каждом из циклов, сам пакет определяет границы выполнения синтаксического анализа. Иными словами, при таких допущениях вычислительная сложность программы P4 линейно зависит от суммарного размера заголовков и никогда не зависит от состояний, собранных в процессе обработки данных (например, от числа потоков или общего числа обработанных пакетов). Эти гарантии необходимы (но не достаточны) для обеспечения быстрой обработки пакетов на разных платформах.

Соответствие платформы языку P4 определяется следующим образом - если конкретная платформа T поддерживает лишь часть языка P4 (например, P4T), программе, написанной на P4T, следует обеспечивать при выполнении поведение, соответствующее описанному в этом документе. Отметим, что совместимые с P4 платформы могут предоставлять произвольные расширения языка P4 и внешние элементы.

3.1. Преимущества P4

По сравнению с современными системами обработки пакетов (например, на основе микрокода в специализированном оборудовании) P4 обеспечивает ряд существенных преимуществ.

- ***Гибкость***. P4 позволяет выразить множество правил пересылки пакетов в форме программ в отличие от традиционных коммутаторов с фиксированными машинами пересылки.
- ***Выразительность***. P4 позволяет выражать изощрённые алгоритмы обработки пакетов, независимые от оборудования, исключительно с помощью операций общего назначения и поиска в таблицах. Такие программы переносимы между платформами, имеющими одинаковую архитектуру (при наличии достаточных ресурсов).
- ***Отображение ресурсов и управлением ими***. Программы P4 описывают ресурсы хранения (например, адрес отправителя IPv4) абстрактно, компиляторы отображают заданные пользователем поля на доступные аппаратные ресурсы и управляют на нижнем уровне такими задачами, как распределение и планирование.
- ***Разработка программ***. Программы P4 предоставляют важные преимущества, такие как проверка типов, сокрытие информации, многократное использование программного кода.
- ***Библиотеки компонент***. Поставляемые производителями библиотеки могут использоваться для включения аппаратно-зависимых функций в переносимые конструкции высокого уровня на языке P4.
- ***Независимое развитие программ и оборудования***. Производители платформ могут применять абстрактную архитектуру для дальнейшего отвязывания низкоуровневых деталей архитектуры от обработки на высоких уровнях.
- ***Отладка***. Производители могут предоставлять программные модели архитектуры для оказания помощи в разработке и отлаживании программ P4.

3.2. Развитие P4 (сравнение с P4 v1.0/v1.1)

По сравнению с $P4_{14}$ (ранняя версия языка) $P4_{16}$ включает множество важных, но несовместимых изменений в синтаксисе и семантике. Развитие от $P4_{14}$ к $P4_{16}$ показано на рисунке 3. В частности, многие функции (включая счётчики, расчёт контрольных сумм, измерители и т. п.) были перенесены из основного языка в библиотеки.

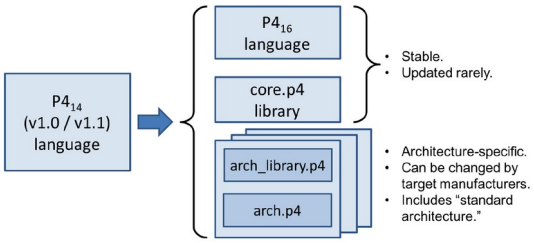

Рисунок 3. Развитие языка P4.

В результате сложный язык (более 70 ключевых слов) стал существенно проще (менее 40 ключевых слов, как представлено в Приложении B) и сопровождается библиотекой базовых конструкций, требуемых для создания программ P4.

В версии 1.1 языка P4 добавлена конструкция extern, которую можно применять для описания библиотечных элементов. Многие конструкции в спецификации v1.1 были преобразованы в библиотечные элементы (включая исключённые из языка конструкции, такие как счётчики и измерители). Ожидается, что некоторые из этих внешних объектов будут стандартизованы и включены потом в новый документ, описывающий библиотечные элементы P4. В этом документе рассматривается несколько примеров внешних конструкций. $P4_{16}$ также вводит и меняет некоторые языковые конструкции v1.1 для описания программируемых частей архитектуры (parser, state, control, package).

Одной из важных целей пересмотра языка $P4_{16}$ было обеспечение стабильного определения. Иными словами, нужно было обеспечить, чтобы все программы, написанные на $P4_{16}$, оставались синтаксически корректными и вели себя одинаково при рассмотрении как программ будущих версий языка. Более того, если какая-то из будущих версий языка откажется от поддержки старых версий, будет возможен простой путь перевода программ $P4_{16}$ на новую версию.

4. Архитектурная модель

Архитектура P4 идентифицирует программные блоки P4 (например, анализаторы, блоки входного и выходного управления потоком, сборщики и т. п.) и их интерфейсы плоскости данных. Архитектуру P4 можно рассматривать как соглашение между программой и целевой платформой. Поэтому каждый производитель должен обеспечить компилятор P4, а также соответствующее определение архитектуры для своей платформы (предполагается что компиляторы P4 могут иметь общий блок предварительной компиляции - front-end для всех вариантов архитектуры). Определение архитектуры не обязательно должно раскрывать всю программируемую часть плоскости данных и

производитель даже может предоставить для своего устройства несколько определений с разными возможностями (например, с поддержкой и без поддержки групповой адресации).

На рисунке 4 показаны интерфейсы плоскости данных между программными блоками P4. Показана платформа с двумя программируемыми блоками (#1 и #2), каждый из которых программируется своим фрагментом кода P4. Интерфейсы платформы с программой P4 организованы через набор регистров управления или сигналов. Входные элементы управления предоставляют информацию программам P4 (например, порт, принявший пакет), а выходные могут быть созданы программами P4 для влияния на поведение платформы (например, выбор выходного порта). Регистры управления (сигналы) представляются в P4 как внутренние метаданные. Программы P4 могут также сохранять данные для каждого пакета и манипулировать ими в качестве пользовательских метаданных.

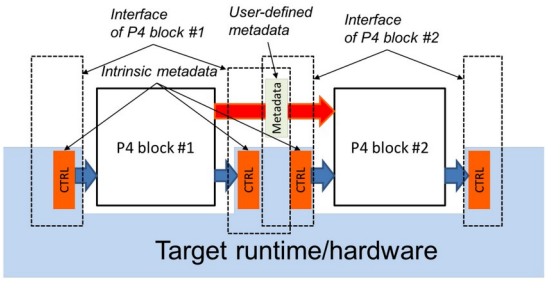

Рисунок 4. Программные интерфейсы P4.

Поведение программы P4 можно полностью описать преобразованиями, отображающими векторы битов на другие векторы битов. Для реальной обработки пакетов архитектурная модель интерпретирует биты, которые программа P4 записывает во внутренние метаданные. Например, для пересылки пакета в конкретный выходной порт программе P4 может потребоваться установить бит drop в другом выделенном регистре управления. Отметим, что детали интерпретации внутренних метаданных зависят от архитектуры.

Программы P4 могут обращаться к службам, реализованным внешними объектами, и функциям, обеспечиваемым архитектурой. На рисунке 5 показана программа P4, обращающаяся к службе встроенного блока платформы для расчёта контрольных сумм. Реализация этого блока не задана в P4, но обеспечивается интерфейс с ним. В общем случае интерфейс для внешнего объекта описывает каждую обеспечиваемую им операцию, а также параметры и возвращаемые типы.

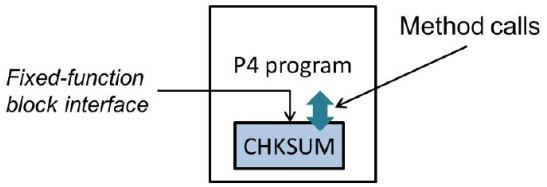

Рисунок 5. Программа P4, вызывающая службы объекта с фиксированными функциями.

В общем случае не предполагается переносимость программ P4 между разными архитектурами. Например, выполнение программы P4, рассылающей широковещательные пакеты путём записи в специальный регистр управления, не будет корректным для платформы, не имеющей такого регистра. Однако программы P4 для определённой архитектуры должны быть переносимыми между всеми платформами, которые точно реализуют соответствующую модель, при наличии достаточных ресурсов.

4.1. Стандартные архитектуры

Предполагается, что сообщество P4 будет развивать небольшой набор стандартных архитектурных моделей, относящихся в конкретным вертикалям. Широкое внедрение таких моделей будет способствовать переносимости программ P4 между разными платформами. Однако определение стандартных архитектур выходит за рамки этого документа.

4.2. Интерфейсы уровня данных

Для описания функционального блока, который можно запрограммировать в P4, архитектура включает объявление типа, задающее интерфейсы между блоком и другими компонентами архитектуры. Например, это может иметь вид

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

Это объявление типа описывает блок с именем MatchActionPipe, который можно запрограммировать с использованием зависящей от данных последовательности вызовов блока СД и других императивных конструкций (указываются ключевым словом control). Интерфейс между блоком MatchActionPipe и другими компонентами архитектуры можно увидеть из этого объявления.

- Первым параметром является 4-битовое значение inputPort. Направление in показывает, что это входной параметр, который не может быть изменён в блоке.
- Вторым параметром является объект типа H с именем parsedHeaders, где H - переменная типа, представляющая заголовки, которые будут далее определены программистом P4. Направление inout показывает, что параметр является сразу входным и выходным.
- Третьим параметром является 4-битовое значение outputPort. Направление out указывает выходной параметр, начальное значение которого не определено, но параметр можно изменять.

4.3. Внешние объекты и функции

Программы P4 могут взаимодействовать с объектами и функциями, обеспечиваемыми архитектурой. Эти объекты описываются с помощью конструкции extern, задающей интерфейсы, открываемые таким объектом уровню данных.

Внешний объект описывает набор реализуемых им методов, но не их реализацию (похоже на абстрактные классы в объектно-ориентированных языках). Например, приведённое ниже описание можно использовать для модуля инкрементного подсчёта контрольной суммы.

```
extern Checksum16 {
        Checksum16();          // конструктор
        void clear();          // подготовка к расчёту
        void update<T>(in T data); // добавление данных в контрольную сумму
        void remove<T>(in T data); // исключение данных из контрольной суммы
        bit<16> get();         // получение контрольной суммы для данных, добавленных после
                    // предшествующего сброса (clear)
}
```

5. Пример очень простого коммутатора

В качестве иллюстрации возможностей архитектуры рассмотрим очень простой коммутатор P4, описав сначала архитектуру, а затем представив полную программу P4, задающую поведение плоскости данных в этом коммутаторе. Пример демонстрирует многие важные возможности языка программирования P4.

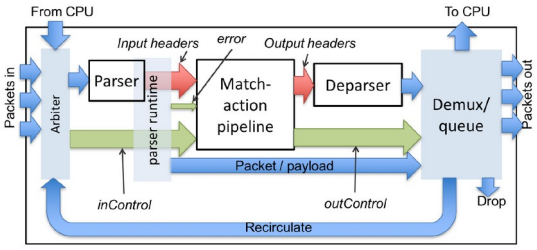

Рисунок 6. Архитектура очень простого коммутатора.

Эта архитектура названа VSS[1] и показана на рисунке 6. В VSS нет ничего особенного - это просто учебный пример, показывающий, как можно описать и запрограммировать коммутатор в P4. VSS имеет множество фиксированных функциональных блоков (голубые прямоугольники), поведение которых описано в параграфе 5.2. Описание архитектуры VSS. Отмеченные белым прямоугольники программируются с использованием P4.

VSS принимает пакеты через один из 8 входных портов Ethernet, канал рециркуляции или порт, подключенный напрямую к CPU. VSS имеет один синтаксический анализатор в единственном конвейере СД, связанном с единственным сборщиком (deparser). После выхода из сборщика пакет передаётся в один из 8 выходных портов Ethernet или в один из трёх специальных портов:

- CPU для передачи плоскости управления;
- Drop в случае отбрасывания пакета;
- Recirculate для повторного прохождения через коммутатор.

Белые прямоугольники на рисунке указывают программируемые блоки и пользователь должен представить программу P4, задающую поведение каждого их этих блоков. Красные стрелки показывают пути заданных пользователем данных, голубые - интерфейсы с фиксированными компонентами, зелёные - интерфейсы плоскости данных для передачи информации между фиксированными и программируемыми блоками (внутренние метаданные программы P4).

5.1. Архитектура VSS

Приведённая ниже программа P4 объявляет VSS, как это мог сделать производитель. Задано несколько типов, констант и 3 программируемых блока, описанных типами и реализованных программой коммутатора.

```
// Файл "very_simple_switch_model.p4"
// Объявление в VSS основной библиотеки P4 требуется
// для определений packet_in и packet_out.
# include <core.p4>
/* Объявления констант и структур */
/* Порты представляются 4-битовыми значениями */
typedef bit<4> PortId;
/* Имеется лишь 8 реальных портов */
const PortId REAL_PORT_COUNT = 4w8;  // 4w8 - 8 в 4-битовом формате
/* Метаданные, сопровождающие входной пакет */
struct InControl {
        PortId inputPort;
}
/* Специальные значения для входных портов */
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;
/* Метаданные, которые должны создаваться для выходных пакетов */
struct OutControl {
        PortId outputPort;
}
/* Специальные значения для выходных портов */
const PortId DROP_PORT = 0xF;
```

[1] *Very Simple Switch - очень простой коммутатор.*

```
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;
/* Прототипы программируемых блоков */
/**
* Программируемый анализатор.
* @param <H> - тип заголовка, задаваемый пользователем
* @param b - входной пакет
* @param parsedHeaders - заголовки, созданные анализатором
*/
parser Parser<H>(packet_in b, out H parsedHeaders);
/**
* Конвейер «сопоставление-действие»
* @param <H> - тип входных и выходных заголовков
* @param headers - заголовки, полученные от анализатора и отправляемые сборщику
* @param parseError - ошибки, которые могут возникать при анализе
* @param inCtrl - информация от архитектуры, сопровождающая входной пакет
* @param outCtrl - информация для архитектуры, сопровождающая выходной пакет
*/
control Pipe<H>(inout H headers,
                in error parseError,  // ошибка анализатора
                in InControl inCtrl,  // входной порт
                out OutControl outCtrl);      // выходной порт
/**
* Сборщик VSS.
* @param <H> - тип заголовков, задаваемый пользователем
* @param b - выходной пакет
* @param outputHeaders - заголовки для выходного пакета
*/
control Deparser<H>(inout H outputHeaders, packet_out b);
/**
* Объявление пакета верхнего уровня. Должен создаваться пользователем.
* Аргументы задают блоки, которые должен создать пользователь.
* @param <H> - заданный пользователем тип обрабатываемых заголовков.
*/
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
// Зависимые от архитектуры объекты, которые могут создаваться.
// Блок контрольных сумм.
extern Checksum16 {
        Checksum16();                 // конструктор
        void clear();                 // подготовка к расчёту
        void update<T>(in T data);    // добавление данных в контрольную сумму
        void remove<T>(in T data);    // исключение данных из контрольной суммы
        bit<16> get();                // контрольная сумма для данных после clear
}
```

Рассмотрим некоторые из элементов.

- Файл core.p4 (Приложение D. Основная библиотека P4) определяет стандартные типы данных и коды ошибок.
- Тип bit<4> указывает строку из 4 битов.
- Синтаксис 4w0xF указывает значение 15, представленное 4 битами. Другим вариантом записи является 4w15. Во многих случаях указатель размера можно опустить, просто написав 15.
- Тип error является встроенным типом P4 для кодов ошибок.

Далее приведено объявление синтаксического анализатора

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

Это объявление описывает интерфейс анализатора, но не содержит его реализации, которую предоставляет программист. Анализатор читает данные из packet_in (предопределённый внешний объект P4 для представления входящего пакета, объявленный в библиотеке core.p4). Анализатор записывает свой вывод (ключевое слово out) в аргумент parsedHeaders. Тип этого аргумента H ещё не известен и будет задан программистом.

Объявление

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

описывает интерфейс конвейера СД (Match-Action) с именем Pipe. Конвейер принимает на входе заголовки headers, параметр для ошибок parseError и данные управления inCtrl. На рисунке 6 показаны источники этих данных. Конвейер записывает свой вывод в outCtrl и должен обновить заголовки, передаваемые сборщику.

Пакет верхнего уровня называется VSS и для программирования VSS пользователь должен создать экземпляр пакета этого типа (см. ниже). Объявление пакета верхнего уровня зависит от типа переменной H

```
package VSS<H>
```

Переменная типа указывает тип, который ещё не известен и должен быть предоставлен пользователем позже. В данном случае H является типом набора заголовков, который пользовательская программа будет обрабатывать. Анализатор будет выдавать разобранное представления этих заголовков, а конвейер СД будет обновлять входные заголовки, заменяя их выходными.

Объявление пакета VSS включает три составных параметра типов Parser, Pipe и Deparser, которые являются объявлениями, описанными здесь. Для программирования платформы нужно представить значения этих параметров.

В этой программе структуры inCtrl и outCtrl представляют регистры управления. Содержимое структур заголовков хранится в регистрах общего назначения. Объявление внешней функции Checksum16 описывает внешний объект, вызываемый для расчёта контрольных сумм.

5.2. Описание архитектуры VSS

Для полного понимания поведения VSS и создания осмысленных программ P4 для неё, а также реализации плоскости управления нужно полностью описать блоки с фиксированной функциональностью. Ниже рассматривается простой пример, иллюстрирующий все детали, которые нужно учитывать при описании архитектуры. Язык P4 не предназначен для описания всех функциональных блоков - он может лишь описать взаимодействие между программируемыми блоками и архитектурой. Для текущей программы этот интерфейс задаётся объявлениями блоков Parser, Pipe и Deparser. На практике предполагается полное описание архитектуры в виде исполняемой программы и/или схем и текста. В этом документе приводится неформальное текстовое описание.

5.2.1. Блок арбитража

Входной блок арбитража выполняет перечисленные ниже функции.

- Приём пакета от одного из физических портов Ethernet, плоскости управления или порта рециркуляции.
- Для пакетов из портов Ethernet рассчитывается и проверяется контрольная сумма трейлера. При несовпадении пакет отбрасывается, при совпадении контрольная сумма удаляется из данных пакета (payload).
- Приём пакета запускает механизм арбитража, если доступно несколько пакетов.
- Если блок арбитража занят обработкой предыдущего пакета и в очереди нет места, входной порт может отбрасывать приходящие пакеты без какого-либо информирования об этом.
- После получения пакета блок арбитража устанавливает значение inCtrl.inputPort, являющееся входным для конвейера СД, и указывает принявший порт пакет. Физические порты Ethernet имеют номера от 0 до 7, порт рециркуляции - 13, а порт CPU - 14.

5.2.2. Блок выполнения синтаксического анализа

Блок выполнения синтаксического анализатора (runtime) работает вместе с анализатором. Блок предоставляет код ошибки конвейеру СД на основе действий анализатора, а также информацию о данных пакета (например, размер оставшейся части данных) блоку демультиплексора. По завершении обработки пакета синтаксическим анализатором вызывается конвейер СД с передачей ему связанных с пакетом метаданных (заголовки и метаданные пользователя).

5.2.3. Блок демультиплексора

Основной функцией блока демультиплексора является получение заголовков для выходных пакетов от сборщика (deparser) и данных пакета от анализатора с целью сборки нового пакета и отправки его в нужный выходной порт. Этот порт указывается значением outCtrl.ouputPort, которое устанавливает конвейер СД.

- Отправка пакета в порт drop вызывает исчезновение пакета.
- Отправка пакета в выходной порт Ethernet 0 - 7 вызывает его передачу через соответствующий физический интерфейс. Пакет может быть помещён в очередь, если выходной интерфейс занят отправкой другого пакета. При отправке интерфейс вычисляет контрольную сумму трейлера Ethernet и помещает её в конец пакета.
- Отправка пакета в выходной порт CPU ведёт к передаче пакета плоскости управления. Процессору передаётся исходный полученный пакет, а не результат работы сборщика (тот просто отбрасывается).
- Отправка пакета в выходной порт рециркуляции передаёт его во входной порт рециркуляции. Это полезно для пакетов, обработку которых невозможно выполнить за один проход.
- Если пакет имеет некорректное значение outputPort (например, 9), он отбрасывается.
- Если блок демультиплексора занят обработкой предыдущего пакета и нет возможности поместить в очередь пакет из сборщика, демультиплексор может отбросить пакет, независимо от указанного выходного порта.

Отметим, что некоторые детали поведения демультиплексора могут показаться неожиданными. Здесь не рассмотрены некоторые важные элементы поведения, связанные с размером очереди, арбитражем и синхронизацией, которые также влияют на обработку пакетов. Стрелка от блока анализатора показывает дополнительный поток информации от анализатора к демультиплексору - обрабатываемый пакет, а также смещение в пакете, где закончена работа анализатора (т. е. начало данных в пакете).

5.2.4. Доступные внешние блоки

Архитектура VSS включает внешний блок инкрементного расчёта контрольных сумм Checksum16, имеющий конструктор и 4 метода:

- clear() готовит блок к новому расчёту;
- update<T>(in T data) добавляет данные в расчёт контрольной суммы (данные должны быть строкой битов, заголовком или структурой с такими значениями), указанные поля добавляются (конкатенация) в порядке их задания в объявлении типа;
- get() возвращает 16-битовую контрольную сумму с дополнением до 1, при вызове этой функции блок контрольных сумм должен получать целое число байтов данных;
- remove<T>(in T data) удаляет из расчёта данные, которые раньше учитывались в контрольной сумме.

5.3. Полная программа VSS

Здесь представлена полна реализация программы P4 для базовой пересылки пакетов IPv4 на платформе VSS. Программа не использует все возможности архитектуры (например, рециркуляцию), по применяет директивы препроцессора #include (6.2. Предварительная обработка).

Синтаксический анализатор пытается распознать заголовок Ethernet, за которым следует заголовок IPv4. Если любой из этих заголовков отсутствует, анализ завершается ошибкой. В остальных случаях информация из заголовков извлекается в структуру Parsed_packet. Конвейер СД, показанный на рисунке 7, включает 4 блока СД (ключевое слово P4 table).

- При возникновении ошибки в анализаторе пакет отбрасывается (outputPort = DROP_PORT).
- Первая таблица использует адрес получателя IPv4 для определения outputPort и адреса IPv4 следующего узла пересылки. Если поиск не дал результата, пакет отбрасывается. Таблица также декрементирует IPv4 ttl.
- Вторая таблица проверяет значение ttl и при обнаружении 0 передаёт пакет плоскости управления через порт CPU.
- Третья таблица использует адрес IPv4 следующего интервала (результат первой таблицы) для определения Ethernet-адреса следующего интервала.
- Последняя таблица использует outputPort для определения Ethernet-адреса (MAC) отправителя в текущем коммутаторе, который помещается в выходной пакет.

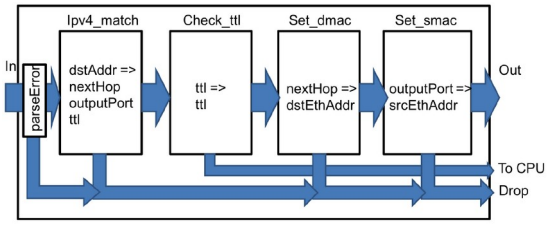

Рисунок 7. Схема конвейера СД, выраженного программой P4 VSS.

Сборщик создаёт выходной пакет, собирая новые заголовки Ethernet и IPv4 по результатам работы конвейера.

```
// Включение основной библиотеки P4
# include <core.p4>
// Включение объявлений архитектуры VSS.
# include "very_simple_switch_model.p4"
// Эта программа обрабатывает пакеты с заголовками Ethernet и IPv4,
// пересылая их по IP-адресу получателя.
typedef bit<48>       EthernetAddress;
typedef bit<32>       IPv4Address;
// Стандартный заголовок Ethernet
header Ethernet_h {
        EthernetAddress dstAddr;
        EthernetAddress srcAddr;
        bit<16> etherType;
}
// заголовок IPv4 (без опций)
header IPv4_h {
        bit<4>          version;
        bit<4>          ihl;
        bit<8>          diffserv;
        bit<16> totalLen;
        bit<16> identification;
        bit<3>          flags;
        bit<13> fragOffset;
        bit<8>          ttl;
        bit<8>          protocol;
        bit<16> hdrChecksum;
        IPv4Address     srcAddr;
        IPv4Address     dstAddr;
}
// Структура проанализированных заголовков
struct Parsed_packet {
        Ethernet_h      ethernet;
        IPv4_h          ip;
}
// Раздел анализатора
// Заданные пользователем ошибки, которые могут возникать при анализе
error {
        IPv4OptionsNotSupported,
        IPv4IncorrectVersion,
        IPv4ChecksumError
}
parser TopParser(packet_in b, out Parsed_packet p) {
Checksum16() ck;        // Создание экземпляра блока контрольных сумм
        state start {
                b.extract(p.ethernet);
                transition select(p.ethernet.etherType) {
                        0x0800: parse_ipv4;
                        // Нет принятого по умолчанию правила — все прочие пакеты отвергаются
```

```
                }
        }
        state parse_ipv4 {
                b.extract(p.ip);
                verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
                verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
                ck.clear();
                ck.update(p.ip);
                // Проверка нулевого значения контрольной суммы
                verify(ck.get() == 16w0, error.IPv4ChecksumError);
                transition accept;
        }
}
// Раздел конвейера CД
control TopPipe(inout Parsed_packet headers,
                        in error parseError, // ошибка анализатора
                        in InControl inCtrl, // входной порт
                        out OutControl outCtrl) {
        IPv4Address nextHop;  // локальная переменная
        /**
        * Указывает отбрасывание пакета установкой выходного порта DROP_PORT
        */
        action Drop_action() {
                outCtrl.outputPort = DROP_PORT;
        }
        /**
        * Установка следующего интервала и выходного порта.
        * Декрементирование поля IPv4 ttl.
        * @param ivp4_dest - адрес IPv4 следующего интервала
        * @param port - выходной порт
        */
        action Set_nhop(IPv4Address ipv4_dest, PortId port) {
                nextHop = ipv4_dest;
                headers.ip.ttl = headers.ip.ttl - 1;
                outCtrl.outputPort = port;
        }
        /**
        * Расчёт адреса следующего интервала IPv4 и выходного порта
        * на основе адреса получателя IPv4 в текущем пакете.
        * Декрементирование поля IPv4 TTL.
        * @param nextHop - адрес IPv4 следующего интервала
        */
        table ipv4_match {
                key = { headers.ip.dstAddr: lpm; }   // самый длинный совпадающий префикс
                actions = {
                        Drop_action;
                        Set_nhop;
                }
                size = 1024;
                default_action = Drop_action;
        }
        /**
        * Передача пакета в порт CPU
        */
        action Send_to_cpu() {
                outCtrl.outputPort = CPU_OUT_PORT;
        }
        /**
        * Проверка TTL и отправка в CPU при значении 0
        */
        table check_ttl {
                key = { headers.ip.ttl: exact; }
                actions = { Send_to_cpu; NoAction; }
                const default_action = NoAction; // определено в core.p4
        }
        /**
        * Установка MAC-адреса получателя пакета
        * @param dmac - MAC-адрес получателя.
        */
        action Set_dmac(EthernetAddress dmac) {
                headers.ethernet.dstAddr = dmac;
        }
        /**
        * Установка адреса Ethernet получателя пакета
        * по IP-адресу следующего интервала.
        * @param nextHop - адрес IPv4 следующего интервала.
        */
        table dmac {
                key = { nextHop: exact; }
                actions = {
                        Drop_action;
                        Set_dmac;
                }
                size = 1024;
```

```
                default_action = Drop_action;
        }
        /**
        * Установка MAC-адреса отправителя.
        * @param smac - MAC-адрес отправителя
        */
        action Set_smac(EthernetAddress smac) {
                headers.ethernet.srcAddr = smac;
        }
        /**
        * Установка MAC-адреса отправителя по выходному порту.
        */
        table smac {
                key = { outCtrl.outputPort: exact; }
                actions = {
                        Drop_action;
                        Set_smac;
                }
                size = 16;
                default_action = Drop_action;
        }
        apply {
                if (parseError != error.NoError) {
                        Drop_action(); // вызов drop напрямую
                        return;
                }
                ipv4_match.apply(); // Результатом совпадений будет переход к nextHop
                if (outCtrl.outputPort == DROP_PORT) return;
                check_ttl.apply();
                if (outCtrl.outputPort == CPU_OUT_PORT) return;
                dmac.apply();
                if (outCtrl.outputPort == DROP_PORT) return;
                smac.apply();
        }
}
// Раздел сборщика
control TopDeparser(inout Parsed_packet p, packet_out b) {
        Checksum16() ck;
        apply {
                b.emit(p.ethernet);
                if (p.ip.isValid()) {
                        ck.clear();                     // Подготовка блока контрольных сумм
                        p.ip.hdrChecksum = 16w0;        // Очистка контрольной суммы
                        ck.update(p.ip);                // Расчёт новой контрольной суммы
                        p.ip.hdrChecksum = ck.get();
                }
                b.emit(p.ip);
        }
}
// Создание экземпляра пакета VSS.
VSS(TopParser(),
        TopPipe(),
        TopDeparser()) main;
```

6. Определение языка P4

Язык P4 можно рассматривать как состоящий из нескольких частей, которые будут далее описаны.

- Базовый язык, включающий типы, переменные, области действия, объявления, операторы, выражения и т. д.
- Субязык для задания анализаторов на основе конечного автомата или машины состояний (12. Анализ пакета).
- Субязык для выражения расчётов, используемых блоками СД, основанными на традиционной императивном потоке управления (13. Блоки управления).
- Субязык описания архитектуры (16. Описание архитектуры).

6.1. Синтаксис и семантика

6.1.1. Грамматика

Полная грамматика $P4_{16}$ описана в Приложении H на основе языка описания грамматики Yacc/Bison, используемого и в данном тексте. В выдержках из грамматики принимается несколько стандартных допущений:

- для обозначения терминальных символов применяются ЗАГЛАВНЫЕ буквы;
- применяется нотация BNF, как показано ниже

```
p4program
        : /* пусто */
        | p4program declaration
        | p4program ';'
        ;
```

Псевдокод, применяемый в основном для описания семантики конструкций P4, выводится фиксированным шрифтом

```
ParserModel.verify(bool condition, error err) {
        if (condition == false) {
```

```
            ParserModel.parserError = err;
            goto reject;
        }
}
```

6.1.2. Семантика и абстрактные машины P4

Семантика P4 описывается в терминах абстрактной машины, выполняющей традиционный императивный код. Для каждого из субязыков P4 (анализаторы, элементы управления) имеется своя абстрактная машина. Для описания этих машин применяется псевдокод и текст.

Компиляторы P4 могут сами организовывать создаваемый код, если видимое извне поведение программ P4 соответствует данной спецификации. Таким внешним поведением считается:

- поведение операций ввода и вывода всех блоков P4;
- состояния, поддерживаемые внешними (extern) блоками.

6.2. Предварительная обработка

Для сборки программ из нескольких исходных файлов P4 компиляторам следует поддерживать некоторые функции препроцессоров C:

- #define для определения макросов (без аргументов);
- #undef;
- #if #else #endif #ifdef #ifndef #elif;
- #include.

Препроцессору также следует удалять символы \ и перевода строки (ASCII 92, 10), используемые для разбиения строк с целью удобочитаемости.

Могут поддерживаться другие возможности препроцессора C (например, макросы с параметрами), но это не гарантируется. Как в C, директива #include может задавать имя файла в угловых скобках или двойных кавычках.

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

Разница заключается в порядке поиска файлов препроцессором при неполном указании пути.

Компиляторам P4 следует корректно обрабатывать директивы #line, которые могут генерироваться в процессе предварительной обработки. Это позволяет собирать программы P4 из множества файлов:

- базовая библиотека P4, определённая в этом документе;
- архитектура, определяющая интерфейсы плоскости данных и блоки extern;
- заданные пользователем библиотеки компонент (например, определения стандартных заголовков);
- программы P4, задающие поведение каждого программируемого блока.

6.2.1. Основная библиотека P4

Спецификация языка P4 определяет основную библиотеку, включающую некоторые базовые программные конструкции. Описание этой библиотеки дано в Приложении D. Все программы P4 должны включать эту библиотеку

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

6.3. Лексические конструкции

Все ключевые слова P4 содержат только символы ASCII, это же должно выполняться для всех идентификаторов P4. Компиляторам P4 следует корректно обрабатывать строки, содержащие 8-битовые символы в комментариях и строковых литералах. Язык P4 различает регистр символов. Пробельные символы, включая новую строку, считаются разделителями. Отступ от начала строки не регламентируется, однако P4 включает блочные конструкции в стиле C во всех примерах этого документа используются отступы в стиле C. Символы табуляции трактуются как пробелы.

Лексер распознает перечисленные ниже терминалы.

- Идентификатор (IDENTIFIER) начинается с буквы или символа подчёркивания и может включать буквы, цифры и символы подчёркивания.
- Идентификатор типа (TYPE_IDENTIFIER) указывает имя типа.
- INTEGER означает целочисленные литералы.
- DONTCARE - одиночный символ подчёркивания.
- Ключевые слова (например, RETURN). По соглашению каждый терминал ключевого слова соответствует ключевому слову языка с таким же произношением, но в символах нижнего регистра. Например, терминал RETURN соответствует ключевому слову return.

6.3.1. Идентификаторы

Идентификаторы P4 могут включать лишь буквы, цифры и символы подчёркивания, а начинаться должны с буквы или подчёркивания. Специальный идентификатор _ зарезервирован для значения, которое «не имеет значения» (don't care), а его тип может зависеть от контекста. Некоторые ключевые слова (например, apply) могут использоваться как идентификаторы, если контекст не позволяет путаницы.

```
nonTypeName
        : IDENTIFIER
```

```
        | APPLY
        | KEY
        | ACTIONS
        | STATE
        | ENTRIES
        | TYPE
        ;

name
        :|;
        nonTypeName
        TYPE_IDENTIFIER
```

6.3.2. Комментарии

P4 поддерживает несколько типов комментариев:

- однострочный комментарий от символов // до конца строки;
- многострочный комментарий между символами /* и */, вложенные многострочные комментарии не поддерживаются;
- комментарии в стиле Javadoc между символами /** и */.

Настоятельно не рекомендуется применять комментарии Javadoc в таблицах и действиях, используемых при создании интерфейса с плоскостью управления.

P4 считает комментарии разделителями и не допускает комментариев внутри маркера (token). Например, bi/**/t будет рассматриваться как два маркера bi и t, а не bit.

6.3.3. Литеральные константы

6.3.3.1. Логические литералы

Для логических (Boolean) значений поддерживаются константы true и false.

6.3.3.2. Целочисленные литералы

Целочисленные литералы являются положительными целыми числами с произвольной разрядностью. По умолчанию литералы считаются десятичными. Для явного указания формата применяются приведённые ниже обозначения:

- 0x или 0X - шестнадцатеричные значения;
- 0o или 0O - восьмеричные значения;
- 0d или 0D - десятичные значения;
- 0b или 0B - двоичные значения.

Размер числового литерала в битах можно задать целым числом без знака с индикатором знака:

- w - целое число без знака;
- s - целое число со знаком.

Отметим, что 0 в начале сам по себе не указывает восьмеричную константу. Символ _ допускается в числовых литералах, но игнорируется при определении значения анализируемого числа. Это позволяет сделать длинные числовые константы более читаемыми. Этот символ не допускается в константах указания размера и не может быть первым символом числового литерала. Внутри литералов не допускаются комментарии и пробелы.

```
32w255                // 32-битовое число без знака со значением 255
32w0d255              // то же самое
32w0xFF         // то же самое
32s0xFF         // 32-битовое число со знаком и значением 255
8w0b10101010          // 8-битовое число без знака со значением 0xAA
8w0b_1010_1010 // то же самое
8w170                 // то же самое
8s0b1010_1010         // 8-битовое число со знаком и значением -86
16w0377         // 16-битовое число без знака со значением 377 (не 255!)
16w0o377              // 16-битовое число без знака со значением 255 (основание 8)
```

6.3.3.3. Строковые литералы

Строковые литералы (строки констант) задаются в форме произвольных последовательностей 8-битовых символов, заключённых в двойные кавычки (", ASCII 34). Строка начинается с символа двойных кавычек и завершается на первом символе двойных кавычек, которому не предшествует нечётное число символов \ (ASCII 92). P4 не проверяет корректность строк (т. е., допустимость использованной кодировки UTF-8).

Поскольку в P4 не поддерживаются операции над строками, строковые литералы, включая завершающие кавычки, обычно пропускаются компилятором P4 без изменений другим сторонним инструментам или backend-компиляторам. Эти инструменты могут по-своему обрабатывать escape-последовательности (например, способ задания символов Unicode или обработку непечатаемых символов ASCII).

Ниже приведено 3 примера строковых литералов.

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

6.4. Соглашения об именовании

P4 поддерживает богатый набор типов. Базовые типы включают битовые строки, числа и ошибки. Имеются также встроенные типы для представления конструкций, таких как анализаторы, конвейеры, действия и таблицы. Пользователь может создавать новые типы на основе структур, перечисляемых значений, заголовков, стеков и объединений заголовков и т. п.

Ниже приведены принятые в документе соглашения об именовании.

- Встроенные типы указываются символами нижнего регистра (например, int<20>).
- Пользовательские типы включают заглавные буквы (например, IPv4Address).
- Переменные типов всегда указываются заглавными буквами (например, parser P<H, IH>()).
- В переменных заглавные буквы не используются (например, ipv4header).
- Константы указываются заглавными буквами (например CPU_PORT).
- Ошибки и перечисляемые указываются в стиле «верблюда» (camel-case), например PacketTooShort.

6.5. Программы P4

Программа P4 представляет собой список объявлений

```
p4program
        : /* пусто */
        | p4program declaration
        | p4program ';'       /* пустое объявление */
        ;
declaration
        : constantDeclaration
        | externDeclaration
        | actionDeclaration
        | parserDeclaration
        | typeDeclaration
        | controlDeclaration
        | instantiation
        | errorDeclaration
        | matchKindDeclaration
        | functionDeclaration
        ;
```

Пустое объявление указывается символом ; (пустые объявления поддерживаются с учётом привычек программистов C/C++ и Java, хотя в некоторых конструкциях, например, struct, точка с запятой в конце не требуется).

6.5.1. Область действия

Некоторые конструкции P4 действуют как пространства имён, создающие локальную область действия имён, включая:

- объявления производных типов (struct, header, header_union, enum) с локальной значимостью имён полей;
- операторы блоков, создающие локальные, лексически замкнутые области действия;
- блоки parser, table, action, control с локальной областью действия;
- объявления с переменными типов, создающими для переменных новые области действия; например, в приведённом ниже объявлении extern область действия типа H завершается в конце определения

```
extern E<H>(/* параметры опущены */) {/* тело опущено */} // область действия H кончается тут.
```

Порядок объявления важен и за исключением состояний анализатора использованию символа должно предшествовать его объявление (это отступление от $P4_{14}$,где объявления разрешаются в любом порядке). Такое требование существенно упрощает реализацию компиляторов P4, позволяя им использовать дополнительную информацию об объявленных идентификаторах при исключении неоднозначностей.

6.5.2. Элементы с состоянием

Большинство конструкций P4 не имеет состояний - результат работы конструкции определяется исключительно входными данными. Имеется лишь две конструкции с состояниями, сохраняющими информацию от пакета к пакету:

- ***таблицы*** для плоскости данных доступны лишь на чтение, но их записи может менять уровень управления;
- ***внешние объекты*** могут иметь состояния, доступные для чтения и записи плоскостям управления и данных; все конструкции из $P4_{14}$, сохраняющие состояние (например, счётчики, измерители, регистры), представлены в $P4_{16}$ объектами extern.

В P4 все элементы с состояниями должны явно выделяться при компиляции путём создания «экземпляров». Кроме того, анализаторы, блоки управления и пакеты могут создавать экземпляры элементов с состояниями. Такие элементы также должны создаваться до использования, даже если они исходно не имеют состояний. Однако таблицы не создаются заранее, хотя у них и есть состояния, - создание экземпляра таблицы происходит при её объявлении. Это сделано для поддержки общего случая, поскольку большинство таблиц используется лишь однократно. Для более точного контроля за состоянием таблиц программистам следует объявлять их в элементах управления.

В примере параграфа 5.3. Полная программа VSS TopParser, TopPipe, TopDeparser, Checksum16 и Switch являются типами. Имеется два экземпляра Checksum16, по одному в TopParser и TopDeparser (обозначены ck). Экземпляры TopParser, TopDeparser, TopPipe и Switch создаются в конце программы в объявлении основного объекта, который является экземпляром типа Switch (пакет).

6.6. Выражения для левой части

Выражениями для левой части (l-value) считаются выражения слева от оператора присваивания или аргументы параметров функций out и inout. Эти значения являются ссылками на хранилище. Корректные варианты приведены ниже.

```
prefixedNonTypeName
        : nonTypeName
        | dotPrefix nonTypeName
        ;
lvalue
        : prefixedNonTypeName
        | lvalue '.' member
        | lvalue '[' expression ']'
        | lvalue '[' expression ':' expression ']'
        ;
```

- Идентификаторы базовых и производных типов.
- Операции доступа к элементам структур, заголовков и объединений заголовков (нотация с точками).
- Ссылки на элементы стека заголовков (8.17. Операции над стеком заголовков) - индексация и ссылки на последний или следующий элемент.
- Результат оператора «нарезки» битов [m:l].

Примером корректного выражения может служить headers.stack[4].field. Отметим, что вызовы методов и функций не могут возвращать l-value.

6.7. Соглашения о вызовах

P4 обеспечивает множество конструкций для создания модульных программ - внешние методы, анализаторы, элементы управления, действия. Все эти конструкции ведут себя подобно процедурам языков программирования общего назначения:

- используются именованные и типизованные параметры;
- создаётся новая локальная область действия для параметров и локальных переменных;
- можно передавать аргументы путём привязки их к параметрам.

Вызовы осуществляются с использованием семантики copy-in/copy-out. Каждый параметр помечается направлением.

- Параметры in доступны лишь для чтения и указание такого параметра в левой части оператора присваивания или передача вызываемому не в качестве аргумента in приведёт к ошибке. Параметры in инициализируются путём копирования значения соответствующего аргумента при выполнении вызова.
- Параметры out за исключением нескольких указанных ниже случаев, инициализируются и трактуются как l-value (6.6. Выражения для левой части) в теле методов и функций. Аргументы, передаваемые как параметры out, должны быть l-value, после выполнения вызова значение параметра копируется в соответствующее место хранилища, выделенное для данного l-value.
- Параметры inout являются входными и выходными сразу (in и out). Аргументы, передаваемые как параметры inout должны быть l-value.
- Отсутствие направления указывает, что параметр соответствует какому-либо из приведённых условий:
  - значение известно в момент компиляции;
  - значение является параметром действия (action), который может быть установлен лишь плоскостью управления;
  - значение является параметром действия, который может быть напрямую установлен другим вызовом (в этом случае поведение аналогично параметрам in).

Параметры out всегда инициализируются в начале выполнения части программы, имеющей такие параметры (например, элемента управления, анализатора, действия, функции и т. п.). Для других направлений такой инициализации нет.

- Если параметр out имеет тип header или header_union, он считается недействительным (invalid).
- Если параметр out имеет тип стека заголовков, все элементы стека считаются недействительными (invalid), а в поле nextIndex устанавливается 0 (8.17. Операции над стеком заголовков).
- Если параметр out имеет композитный тип (например, struct), отличающийся от перечисленных выше, правила применяются рекурсивно к элементам композитного типа.
- Если параметр out имеет иной тип (например, bit<W>), реализация не обязана инициализировать его предсказуемым значением.

Например, если параметр out имеет тип s2_t и имя p

```
header h1_t {
        bit<8> f1;
        bit<8> f2;
}
struct s1_t {
        h1_t h1a;
        bit<3> a;
        bit<7> b;
```

```
}
struct s2_t {
        h1_t h1b;
        s1_t s1;
        bit<5> c;
}
```

тогда в начале выполнения части программы, имеющей выходной параметр p, он должен инициализироваться с объявлением p.h1b и p.s1.h1a недействительными. Остальные части p инициализировать не требуется. Аргументы оцениваются слева направо до вызова самой функции. Порядок оценки важен, когда представленное для аргумента выражение может давать побочные эффекты. Например,

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

Отметим, что оценка g может изменить аргумент a, поэтому компилятор должен убедиться, что значение, переданное f в качестве первого параметра, не было изменено при оценке второго аргумента. Семантика оценки для вызовов функций задаётся приведённым ниже алгоритмом (реализация может менять его при условии сохранения результата).

1. Аргументы оцениваются слева направо в соответствии с выражением при вызове функции.
2. Если параметр имеет принятое по умолчанию значение и соответствующий аргумент не представлен, в качестве аргумента применяется подразумеваемое значение.
3. Для каждого аргумента out и inout сохраняется соответствующее значение l-value (это не позволяет изменить его при оценке последующих аргументов). Это важно, если аргумент содержит операции индексирования в стек заголовков.
4. Значения каждого аргумента сохраняется во временной области.
5. Функция вызывается с аргументами из временной области. Эти аргументы никогда не являются псевдонимами друг друга, поэтому такой «сгенерированный» вызов функции можно реализовать с помощью ссылки (call-by-reference), если архитектура это позволяет.
6. При возврате из функции временные значения, соответствующие аргументам out и inout копируются слева направо в l-value, сохранённые в п. 2.

В соответствии с этим алгоритмом приведённый выше вызов функции эквивалентен последовательности операторов

```
bit tmp1 = a;          // оценка a и сохранение результата
bit tmp2 = g(a);       // оценка g(a), сохранение результата, изменение a
f(tmp1, tmp2);         // оценка f, изменение tmp1
a = tmp1;              // копирование результата inout обратно в a
```

Для подчёркивания важности п. 2 приведённого выше алгоритма рассмотрим пример

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

Оценка этого вызова эквивалента последовательности операторов

```
bit tmp1 = a;          // сохранение a
bit tmp2 = s[tmp1].z; // оценка первого аргумента
bit tmp3 = g(a);       // оценка второго аргумента, изменение a
f(tmp2, tmp3);         // оценка f, изменение tmp2
s[tmp1].z = tmp2;      // копирование результата inout обратно, не в s[a].z
```

При использовании объектов extern в качестве аргументов их можно передавать лишь без направления (см., например, аргументы packet в примере VSS).

6.7.1. Обоснование

Основная причина использования семантики copy-in/copy-out (вместо традиционной call-by-reference) заключается в контроле побочных эффектов внешних функций и методов, которые являются основным механизмом взаимодействия программы P4 со своим окружением. Семантика copy-in/copy-out не даёт внешним функциям удерживать ссылки на объекты программы P4 и это позволяет компилятору ограничить побочные влияния внешних функций на программу P4 как в пространстве (влияние лишь на параметры out), так и во времени (влияние лишь при вызове функции).

В общем случае внешние функции могут делать все, что угодно - хранить информацию в глобальном хранилище, порождать отдельные потоки, «вступать в сговор» с другими для совместного использования информации, - но они не имеют доступа к переменным программы P4. Семантика copy-in/copy-out позволяет компилятору считать программу P4 вызывающей внешние функции.

Имеется ещё два преимущества использования семантики copy-in/copy-out:

- возможность компилировать программы P4 для архитектуры, не поддерживающей ссылки (например, при размещении данных в именованных регистрах), которая может требовать индексов в стеки заголовков, появляющиеся в программе, чтобы получить значения во время компиляции;
- упрощение анализа в компиляторе, поскольку параметры функций не могут быть псевдонимами друг друга в теле функций.

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

nonEmptyParameterList
```

```
        : parameter
        | nonEmptyParameterList ',' parameter
        ;

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

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

Ниже кратко перечислены ограничения, связанные с направлением параметров.

- При использовании в качестве аргументов внешние объекты должны передаваться без направления.
- Все параметры конструкторов оцениваются во время компиляции, поэтому они не могут иметь направления, это относится к объектам package, control, parser, extern. Значения таких параметров должны быть заданы в момент компиляции и обеспечивать возможность оценки при компиляции (14. Параметризация).
- Для действий все параметры без направления должны быть в конце списка параметров. Для действий в таблице должны указываться лишь параметры с направлением (13.1. Действия).
- Действия могут также вызываться явно с использованием синтаксиса функций из блока управления или другого действия. При этом значения всех параметров действия должны быть заданы явно, включая значения параметров без направления, которые в такой ситуации ведут себя как параметры in (13.1.1. Вызов действия).
- Принятые по умолчанию значение разрешены лишь для параметров без направления и in, они должны преобразовываться в константы, доступные при компиляции.

6.7.2. Необязательные параметры

Параметр, аннотированный как @optional, является необязательным и пользователь может опустить его значение при вызове. Необязательные параметры могут присутствовать лишь в аргументах пакетов, внешних функций и методов, а также конструкторов объектов extern. Такие параметры не могут иметь принятых по умолчанию значений. Если конструкция, подобная процедуре, использует необязательные параметры и подразумеваемые значения, она может вызываться лишь с указанием именованных аргументов. Рекомендуется (но не требуется) размещать необязательные параметры в конце списка параметров.

Реализация объектов с необязательными параметрами не задана в P4, поэтому назначение и реализация таких объектов должны задаваться целевой архитектурой. Например, можно представить архитектуру двухэтапной коммутации с необязательным вторым этапом. Это можно объявить как пакет с необязательным параметром

```
package pipeline(/* параметры опущены */);
package switch(pipeline first, @optional pipeline second);

pipeline(/* аргументы опущены */) ingress;
switch(ingress) main; // коммутатор с одноэтапным конвейером
```

Здесь целевая архитектура может реализовать необязательный аргумент, используя пустой конвейер. Ниже приведён пример с необязательными параметрами и параметрами с подразумеваемыми значениями.

```
extern void h(in bit<32> a, in bool b = true);      // принятое по умолчанию значение
// вызовы функций
h(10);          // то же, что и h(10, true);
h(a = 10);      // то же, что и h(10, true);
h(a = 10, b = true);
struct Empty {}
control nothing(inout Empty h, inout Empty m) {
        apply {}
}
parser parserProto<H, M>(packet_in p, out H h, inout M m);
control controlProto<H, M>(inout H h, inout M m);

package pack<HP, MP, HC, MC>(@optional parserProto<HP, MP> _parser, // необязательный параметр
                     controlProto<HC, MC> _control = nothing()); // подразумеваемое значение

pack() main;   // Нет значения _parser, а _control является экземпляром nothing()
```

6.8. Распознавание имён

Объекты P4, создающие пространства имён, организованы в иерархию. Имеется также безымянное пространство верхнего уровня, содержащее все объявления верхнего уровня. Идентификаторы, начинающиеся с точки (.), всегда относятся к пространству имён верхнего уровня.

```
const bit<32> x = 2;
control c() {
        int<32> x = 0;
        apply {
                x = x + (int<32>).x;   // x - локальная переменная int<32>,
                                       // .x - переменная bit<32> верхнего уровня
        }
}
```

Распознавание идентификаторов по ссылкам предпринимается изнутри, начиная с текущей области действия и заканчивая лексически замкнутыми областями. Компилятор может выдавать предупреждения при возможности неоднозначного распознавания имени (затенение имён).

```
const bit<4> x = 1;
control p() {
        const bit<8> x = 8;    // объявление x затеняет глобальную переменную x
        const bit<4> y = .x;   // ссылка на x верхнего уровня
        const bit<8> z = x;    // ссылка на локальную (p) переменную x
        apply {}
}
```

6.9. Видимость

Идентификаторы, определённые на верхнем уровне, видны глобально. Объявления внутри анализатора или элемента управления являются приватными и не могут упоминаться за пределами этого объекта.

7. Типы данных P4

$P4_{16}$ является статически типизованным. Программы, не прошедшие проверку типов, считаются недействительными и отвергаются компилятором. P4 поддерживает множество базовых типов, а также операторы для создания производных типов. Некоторые значения допускают приведение к другому типу, однако неявное приведение допускается лишь в некоторых ситуациях, а диапазон доступных приведений намеренно ограничен.

7.1. Базовые типы

Ниже перечислены встроенные базовые типы, поддерживаемые P4:

- void не имеет значения и используется лишь в редких ограниченных обстоятельствах;
- error служит для передачи сведений об ошибках независимым от платформы, управляемым компилятором способом;
- string может применяться лишь для строковых констант во время компиляции;
- match_kind служит для описания реализации поиска в таблицах;
- bool представляет логические значения;
- int служит для представления целых чисел произвольного размера;
- bit<> - битовые строки фиксированного размера;
- int<> - целые числа фиксированного размера, представленные дополнением до 2;
- varbit<> - битовые строки с ограничением максимального размера.

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

7.1.1. Тип void

Тип указывается ключевым словом void и не содержит значений. Тип не указан в приведённом выше правиле baseType, поскольку его использование в программах P4 существенно ограничено.

7.1.2. Тип error

Тип error содержит значения (opaque), которые могут служить для сигнализации ошибок. Константы типа error определяются в форме

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

Константы error помещаются в пространство имён ошибок, независимо от места их определения. Тип error похож на тип enum в других языках. Программа может включать множество объявлений ошибок, которые компилятор собирает вместе. Объявление одной константы error несколько раз является ошибкой. Выражения типа error описаны в параграфе 8.2. Операции над типом error. Например, ниже приведено определение двух констант (из основной библиотеки P4).

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

Базовое представление ошибок зависит от платформы.

7.1.3. Тип match_kind

Тип match_kind очень похож на error и применяется для объявления набора имён, которые могут служить свойствами ключа таблицы (13.2.1. Свойства таблицы). Все идентификаторы помещаются в пространство имён верхнего уровня. Определение одного идентификатора match_kind несколько раз является ошибкой.

```
matchKindDeclaration
        : MATCH_KIND '{' identifierList '}'
        ;
```

Основная библиотека P4 содержит приведённое ниже объявление match_kind.

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

Архитектура может не поддерживать дополнительные типы match_kind. Объявление новых match_kind может выполняться лишь в файлах определения моделей и программисты P4 не могут делать таких объявлений.

7.1.4. Логический тип

Логический тип bool имеет значения false и true, не относящиеся к integer или bit-string.

7.1.5. Строки

Для представления строк используется тип string. Операций с этим типом в языке не выполняется и невозможно объявить переменную типа string. Параметры этого типа не могут иметь направления (6.7. Соглашения о вызовах). P4 не поддерживает манипуляций со строками в плоскости данных и этот тип можно применять лишь для констант, известных в момент компиляции. Это может быть полезно, например, для платформ, поддерживающих внешнюю функцию для регистрации событий вида

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

В программах P4 могут использоваться лишь строковые константы, описанные в параграфе 6.3.3.3. Строковые литералы. Например, приведённая ниже аннотация указывает, что заданное имя следует применять для таблицы при генерации API плоскости управления.

```
@name("acl") table t1 { /* тело опущено */ }
```

7.1.6. Целые числа

P4 поддерживает целые числа произвольного размера. Особенности типизации целых чисел приведены ниже.

- Типизация целых чисел поддерживает произвольный фиксированный размер чисел. В частности тип результата выражения зависит лишь от операндов, а не от использования результата.
- P4 пытается преодолеть многие особенности поведения C, включающего размер целых чисел (int), в результате чего комбинации целочисленных типов не приводят к неопределённому поведению.
- Правила типизации P4 выбраны так, чтобы поведение было похоже на традиционные программы C.
- Вместо правил, ведущих к неожиданным результатам (например, сравнение в C чисел со знаком и без знака), здесь запрещены выражения с неоднозначным результатом. Например, P4 не разрешает двоичные операции, комбинирующие числа со знаком и без знака.

Приоритет арифметических операций идентичен принятому в C.

7.1.6.1. Переносимость

Ни одна платформа P4 не может поддерживать все возможные типы и операции. Например, тип bit<23132312> корректен в P4, но с большой вероятностью не будет поддерживаться на практике всеми платформами. Поэтому каждая платформа вносит ограничения в число поддерживаемых типов. Такие ограничения могут включать:

- максимальный размер (ширина);
- требования к выравниванию и заполнению (например, могут поддерживаться лишь арифметические операции для целых байтов);
- ограничения для некоторых операндов (например, на значения сомножителей или величины сдвига).

В документацию целевых платформ следует включать такие ограничения, а нацеленным на платформы компиляторам следует выдавать сообщения об ошибках при выходе за пределы ограничений. Архитектура может отвергать корректные программы P4, сохраняя совместимость со спецификацией P4. Однако, если архитектура считает программу P4 пригодной, поведение такой программы должно соответствовать данной спецификации.

7.1.6.2. Целые числа без знака (bit-string)

Целые числа без знака (bit-string) могут иметь произвольный размер в битах. Строка битов размера W обозначается bit<W>. Значение W должно быть известно (вычисляемо) в момент компиляции (17.1. Известные при компиляции значения) и быть целым числом больше 0. Выражения для размера числа должны указываться в скобках.

```
const bit<32> x = 10;          // 32-битовая константа со значением 10.
const bit<(x + 2)> y = 15;     // 12-битовая константа со значением 15.
                               // при указании размера нужно использовать скобки ()
```

Биты в bit-string нумеруются от 0 до W-1, бит 0 является младшим, W-1 - старшим. Например, тип bit<128> указывает битовые строки размером 128 битов с номерами битов от 0 до 127, где бит 127 является старшим. Тип bit является сокращением для bit<1>.

Архитектура P4 может вносить дополнительные ограничения для битовых типов, например, может ограничиваться максимальный размер или поддерживаться некоторые арифметические операции лишь для определённых размеров (скажем, 16, 32, 64 бита). Операции, доступные для целых чисел без знака описаны в параграфе 8.5. Операции над битовыми типами (целые числа без знака).

7.1.6.3. Целые числа со знаком

Целые числа со знаком представляются дополнением до 2. Целое число размером W битов объявляется в форме int<W>, где W должно быть выражением, которое в момент компиляции преобразуется в положительное целое число. Биты числа нумеруются от 0 до W-1, бит 0 является младшим, а бит W-1 содержит знак. Например, тип int<64> описывает целые числа, представляемые 64 битами с номерами от 0 до 63, где бит 63 (старший) задаёт знак.

Архитектура P4 может вносить дополнительные ограничения для чисел со знаком, например, может ограничиваться максимальный размер или поддерживаться некоторые арифметические операции лишь для определённых размеров (скажем, 16, 32, 64 бита). Операции над целыми числами со знаком описаны в параграфе 8.6. Операции над целыми числами фиксированного размера со знаком.

Целое число со знаком размера 1 int<1> может иметь два корректных значения - 0 и -1.

7.1.6.4. Динамические строки битов

В некоторых сетевых протоколах используются поля, размер которых становится известен лишь в процессе работы (например, опции IPv4). Для поддержки ограниченных манипуляций с такими полями в P4 применяется специальный тип битовых строк, размер которых задаётся в процессе работы - varbit. Тип varbit<W> указывает строку битов размером не более W, где W - положительное целое число, известное при компиляции. Например, тип varbit<120> означает строки битов размером от 0 до 120. Большинство операций, применимых к битовым строкам фиксированного размера (целые числа без знака), не может быть выполнено для динамических битовых строк.

Архитектура P4 может вносить дополнительные ограничения для типа varbit, например, может ограничиваться максимальный размер или вводится требование использовать при работе лишь значения varbit с целым числом байтов. Операции над varbit описаны в параграфе 8.8. Операции над битовыми строками переменного размера.

7.1.6.5. Целые числа “бесконечной точности”

Тип данных «бесконечной точности» описывает целые числа неограниченного размера - int. Этот тип зарезервирован для целочисленных литералов и выражений, включающих лишь литералы. Ни одно значение P4 не может иметь тип int в процессе работы - компилятор преобразует значения int в подходящие типы фиксированных размеров.

Операции над типом int описаны в параграфе 8.7. Операции над целыми числами произвольной точности. Приведённый ниже пример показывает определения трёх констант типа int.

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

7.1.6.6. Целочисленные литералы

Типы целочисленных литералов (констант) включают:

- простые целочисленные константы типа int;
- положительные целые числа с префиксом размера N и символом w типа bit<N>.
- целые числа с префиксом размера N и символом s типа int<N>.

В таблице показано несколько примеров целочисленных литералов с их типами. Дополнительные примеры даны в параграфе 6.3.3. Литеральные константы.

Литерал	***Интерпретация***
10	Тип int, значение 10
8w10	Тип bit<8>, значение 10
8s10	Тип int<8>, значение 10
2s3	Тип int<2>, значение -1 (последние 2 бита), предупреждение о переполнении
1w10	Тип bit<1>, значение 0 (последний бит), предупреждение о переполнении
1s1	Тип int<1>, значение -1, предупреждение о переполнении

7.2. Производные типы

P4 поддерживает множество конструкторов типов, которые можно применять для создания производных типов:

- enum;
- header;
- стек заголовков;
- struct;
- header_union;
- tuple;
- специализация типа;
- extern;
- parser;
- control;
- package.

Типы header, header_union, enum, struct, extern, parser, control, package могут использоваться лишь в объявлениях типа, где они задают новое имя типа, которое впоследствии позволяет указывать этот тип.

Другие типы не могут быть объявлены, но синтезируются компилятором для представления типов некоторых языковых конструкций. Эти типы описаны в параграфе 7.2.8. Синтезируемые типы данных (set и function). Например, программист не может объявить переменную типа set, но можно задать выражение, которое будет преобразовываться в этот тип. Эти типы используются в процессе проверки типов.

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

derivedTypeDeclaration
        : headerTypeDeclaration
        | headerUnionDeclaration
        | structTypeDeclaration
        | enumDeclaration
        ;

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

namedType
        : typeName
        | specializedType
        ;

prefixedType
        : TYPE_IDENTIFIER
        | dotPrefix TYPE_IDENTIFIER
        ;

typeName
        : prefixedType
        ;
```

7.2.1. Перечисляемые типы

Перечисляемый тип определяется с использованием показанного ниже синтаксиса.

```
enumDeclaration
        : optAnnotations ENUM name '{' identifierList '}'
        | optAnnotations ENUM BIT '<' INTEGER '>' name '{' specifiedIdentifierList '}'
        ;

identifierList
        : name
        | identifierList ',' name
        ;

specifiedIdentifierList
        : specifiedIdentifier
        | specifiedIdentifierList ',' specifiedIdentifier
        ;

specifiedIdentifier
        : name '=' initializer
        ;
```

Например, объявление

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

вводит новый перечисляемый тип, включающий 4 константы (например, Suits.Clubs). Объявление enum вводит новый идентификатор в текущей области действия для именования созданного типа. Базовое представление перечисляемого типа Suits не задано, поэтому его «размер» в битах не задаётся (зависит от платформы).

Можно также задать enum с базовым представлением. Иногда это называют сериализацией enum, поскольку в заголовки можно включать поля данного типа enum. Это требует от программиста предоставить целочисленный тип фиксированного размера без знака или со знаком и связанное с ним значение для каждого символьного имени в enum. Идентификатор typeRef (см. выше) должен указывать один из следующих типов:

- целое число без знака, т. е. bit<W> с неким постоянным W;
- целое число со знаком, т. е. int<W> с неким постоянным W;
- заданное в typedef имя типа, для которого базовым служит один из указанных выше типов, или другое имя typedef, удовлетворяющее этим условиям.

Например, объявление

```
enum bit<16> EtherType {
        VLAN    = 0x8100,
        QINQ    = 0x9100,
```

```
        MPLS    = 0x8847,
        IPV4    = 0x0800,
        IPV6    = 0x86dd
}
```

вводит новый перечисляемый тип с 5 константами (например, EtherType.IPV4). Это объявление enum задаёт представления целым числом фиксированного размера без знака для каждого элемента enum и указывает базовый тип bit<16>. Этот тип объявления enum можно считать объявлением нового типа bit<16>, где переменные или поля этого типа выражаются 16-битовыми целыми числами без знака и отображения символьных имён на числа, заданные enum, по сути определяют константы. Таким способом enum с базовым типом можно представлять как тип, выведенный из базового, что позволяет использовать равенство, присвоение и приведение к базовому типу или из него.

Предполагается, что реализация компилятора будет выдавать ошибку, если целочисленное представление фиксированного размера для enum будет выходить за пределы базового типа. Например, объявление

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

будет вызывать ошибку, поскольку значение 300, связанное с FailingExample.unrepresentable выходит за пределы bit<8>.

Выражение для инициализации (initializer) должно быть известно в момент компиляции. Аннотации, представляемые нетерминальными optAnnotations, описаны в разделе 18. Аннотации, операции со значениями enum - в параграфе 8.3. Операции над типом enum.

7.2.2. Типы заголовков

Для определения типа header применяется показанный ниже синтаксис.

```
headerTypeDeclaration
        : optAnnotations HEADER name '{' structFieldList '}'
        ;

structFieldList
        : /* пусто */
        | structFieldList structField
        ;

structField
        : optAnnotations typeRef name ';'
        ;
```

где каждое в качестве typeRef можно применять строки битов (переменного или фиксированного размера), целые числами фиксированного размера со знаком, логический тип или struct (с полями типа struct) с произвольной вложенностью, если все типы «листьев» относятся к bit<W>, int<W>, сериализуемым enum или bool. При использовании bool внутри заголовка P4, все реализации кодируют bool однобитовым полем, где 1 представляет true, а 0 - false.

Объявление заголовка вводит новый идентификатор в текущей области действия, который можно указывать заданным идентификатором типа. Заголовок похож на структуру C, содержащую все указанные поля. Однако заголовок также включает скрытое логическое поле validity. Когда это поле установлено (true), заголовок считается действительным. При объявлении локальной переменной типа header для бита validity автоматически устанавливается значение false. Менять этот бит можно с использованием методов isValid(), setValid() и setInvalid(), как описано в параграфе 8.16. Операции над заголовками.

Отметим, что вложенные заголовки не поддерживаются. Одна из сложностей состоит в усложнении определения поведения произвольных последовательностей операций setValid, setInvalid и emit. Рассмотрим пример, где заголовок h1 содержит заголовок h2 и оба заголовка действительны (valid). Программа выполняет h2.setInvalid(), за которым следует packet.emit(h1). Следует ли emit выводить все поля h1, пропуская h2? Следует ли h1.setInvalid() объявлять недействительными все заголовки внутри h1 независимо от глубины вложенности?

Заголовок может быть пустым, как показано ниже.

```
header Empty_h { }
```

Отметим, что бит validity имеется и в пустом заголовке.

При наличии struct внутри header порядок полей при извлечении и вызове emit совпадает с порядком, указанным в исходном коде. Ниже приведён пример заголовка, включающего struct.

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

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

```
        ipv6_addr       dst;
}
```

Заголовки без полей varbit имеют фиксированный размер, заголовки с полями varbit - переменный. Размер фиксированного заголовка (в битах) не меняется и равен сумме размеров всех его полей (без учёта бита validity). Для полей заголовка не используется заполнение или выравнивание. Платформы могут вносить свои ограничения для типа header, например, ограничивать размер целым числом байтов. Ниже приведён пример определения для типичного заголовка Ethernet.

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

Приведённое ниже объявление переменной использует определение типа Ethernet_h

```
Ethernet_h ethernetHeader;
```

Язык анализа P4 предоставляет точный метод извлечения, который можно применять для «заполнения» полей header из заголовков пакета, как описано в параграфе 12.8. Извлечение данных. При успешном извлечении полей для бита validity устанавливается значение true. Ниже приведён пример заголовка IPv4 с опциями переменного размера.

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

Как показано в примере параграфа 12.8.2. Извлечение при переменном размере, другим способом поддержки заголовков с полями переменного размера является определение двух заголовков - фиксированного и с полем varbit, поля из которых извлекаются раздельно.

7.2.3. Стеки заголовков

Стек заголовков представляет собой массив заголовков, определяемый как

```
headerStackType
        : typeName '[' expression ']'
        ;
```

Здесь typeName указывает имя заголовка. Для стека заголовков hs[n] значение n указывает максимальный размер и должно быть положительным целым числом, значение которого известно в момент компиляции. Вложенные стеки заголовков не поддерживаются. Во время работы стек содержит n заголовков типа typeName, из которых действительна лишь часть. Выражения со стеками заголовков рассмотрены в параграфе 8.17. Операции над стеком заголовков.

Например, объявление

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

вводит стек заголовков с именем mpls, содержащий 10 элементов типа Mpls_h.

7.2.4. Объединения заголовков

Объединение заголовков представляет собой другой вариант структуры из нескольких заголовков. Объединение можно использовать для представления опций в таких протоколах, как TCP и IP. Объединение также подсказывает компилятору P4, что может присутствовать лишь один вариант, позволяя экономить пространство хранилища.

```
headerUnionDeclaration
        : optAnnotations HEADER_UNION name
                '{' structFieldList '}'
        ;
```

Это объявление вводит новый тип с указанным именем в текущей области действия. Каждый элемент из списка полей, используемых для объявления объединения заголовков, должен иметь тип header. Однако можно указать пустой список полей.

Представленный ниже тип Ip_h представляет объединение заголовков IPv4 и IPv6.

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

Объединение заголовков не рассматривается как тип с фиксированным размером.

7.2.5. Структурные типы

Структурные типы P4 определяются, как показано ниже.

```
structTypeDeclaration
        : optAnnotations STRUCT name '{' structFieldList '}'
        ;
```

Объявление вводит новый тип с указанным именем в текущей области действия. Допускаются пустые структуры. Например, приведённая ниже структура Parsed_headers содержит заголовки, распознаваемые простым анализатором.

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

7.2.6. Кортежи

Кортежи (tuple) похожи на структуры в том, что могут включать много значений. Однако поля кортежей не имеют имён. Тип кортежа с n компонентами типов T1,...,Tn определяется в форме

```
tuple<T1,/* поля опущены */,Tn>

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

Операторы для работы с tuple описаны в параграфах 8.10. Операции над кортежами и 8.11. Операции над списками.

Тип tuple<> задаёт кортеж без компонентов.

7.2.7. Правила вложенности типов

В таблице указаны все типы, которые могут служить элементами заголовков, объединений заголовков, структур или кортежей. Отметим, что int указывает целое число «бесконечной» точности, размер которого не задан.

	Тип контейнера		
Тип элемента	***header***	***header_union***	***struct или tuple***
bit<W>	разрешено	ошибка	разрешено
int<W>	разрешено	ошибка	разрешено
varbit<W>	разрешено	ошибка	разрешено
int	ошибка	ошибка	ошибка
void	ошибка	ошибка	ошибка
error	ошибка	ошибка	разрешено
match_kind	ошибка	ошибка	ошибка
bool	разрешено	ошибка	разрешено
enum	разрешено	ошибка	разрешено
header	ошибка	разрешено	разрешено
header stack	ошибка	ошибка	разрешено
header_union	ошибка	ошибка	разрешено
struct	разрешено	ошибка	разрешено
tuple	ошибка	ошибка	разрешено

Тип int не имеет точных требований к хранилищу в отличие от bit<> и int<>. Значения match_kind бесполезно сохранять в переменной, поскольку они служат лишь для задания сопоставлений полей с ключами поиска в таблице, объявляемых во время компиляции. Тип void бесполезен как часть структуры данных. Заголовки должны иметь точно заданные форматы в виде последовательности битов, чтобы их можно было анализировать и собирать заново.

Отметим, что двухаргументный метод extract (12.8.2. Извлечение при переменном размере) для пакетов поддерживает в заголовке лишь одно поле varbit.

Ниже перечислены все типы, которые могут присутствовать в качестве базовых в typedef или объявлениях типа.

Базовый тип B	***typedef B <name>***	***type B <name>***
bit<W>	разрешено	разрешено
int<W>	разрешено	разрешено
varbit<W>	разрешено	ошибка
int	разрешено	ошибка
void	ошибка	ошибка
error	разрешено	ошибка
match_kind	ошибка	ошибка
bool	разрешено	разрешено
enum	разрешено	ошибка
header	разрешено	ошибка
header stack	разрешено	ошибка
header_union	разрешено	ошибка
struct	разрешено	ошибка
tuple	разрешено	ошибка
Имя typedef	разрешено	разрешено
Имя type	разрешено	разрешено

7.2.8. Синтезируемые типы данных

Для проверки типов компилятор P4 может синтезировать представления некоторых типов, которые пользователь не может выразить напрямую. К таким типам относятся set и function.

7.2.8.1. Тип set

Тип set<T> описывает набор значений типа T и может применяться в ограниченном контексте P4. Например, выражение для диапазона 8w5 .. 8w8 описывает набор 8-битовых чисел 5, 6, 7 и 8, т. е. они имеют тип set<bit<8>>;. Это выражение можно использовать как метку в выражении select (12.6. Выражения для выбора), соответствующую любому числу из диапазона. Типы set не могут быть именованными или объявленными программистом P4, они лишь синтезируются компилятором для проверки типов. Выражения типа set описаны в параграфе 8.13. Операции над set.

7.2.8.2. Тип function

Типы function создаются компилятором P4 для представления функций (явных или внешних) и методов при проверке типов. Тип функции называют также её сигнатурой. Библиотеки могут включать объявления функций и внешних функций. Ниже приведены два примера определений.

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

bit<32> add(in bit<32> left, in bit<32> right) {
        return left + right;
}
```

Эти определения описывают два объекта:

- random, который является типом function и представляет:
  - функция имеет результат типа void;
  - функция принимает два параметра;
  - первый параметр имеет направление in, тип bit<5> и имя logRange;
  - второй параметр имеет направление out, тип bit<32> и имя value;
- add также является типом function и представляет
  - результат типа bit<32>;
  - функция принимает два параметра;
  - оба параметра имеют направление in и тип bit<32>.

7.2.9. Внешние типы

P4 поддерживает объявления внешних объектов и внешних функций, как показано ниже.

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

7.2.9.1. Внешние функции

Объявление внешней функции задаёт имя и сигнатуру типа для функции, не задавая её реализации.

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

Пример объявления внешней функции приведён в параграфе 7.2.8.2. Тип function.

7.2.9.2. Внешние объекты

Объявление объекта extern указывает объект и все методы, которые могут быть вызваны для выполнения расчётов и смены состояния объекта. Объявления внешних объектов могут также включать метод конструктора, имя которого должно совпадать с именем содержащего эго типа extern, но не иметь параметров типа и возврата. Объявления внешних объектов могут присутствовать лишь в местах, разрешённых архитектурной моделью, и это может зависеть от платформы.

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

methodPrototype
        : optAnnotations functionPrototype ';'
        | optAnnotations TYPE_IDENTIFIER '(' parameterList ')' ';' // конструктор
        ;

typeOrVoid
        : typeRef
        | VOID
        | IDENTIFIER   // может быть переменной типа
        ;

optTypeParameters
        : /* пусто */
        | '<' typeParameterList '>'
        ;
```

```
typeParameterList
        : name
        | typeParameterList ',' name
        ;
```

Например, в основной библиотеке P4 вводятся внешние объекты packet_in и packet_out для операций над пакетами (12.8. Извлечение данных и 15. Сборка пакета). Ниже приведён пример, показывающий вызов этих методов для пакета.

```
extern packet_out {
        void emit<T>(in T hdr);
}
control d(packet_out b, in Hdr h) {
        apply {
                b.emit(h.ipv4);         // Запись заголовка IPv4 в выходной пакет вызовом emit
}
```

В P4 «форсирование» поддерживается лишь для функций и методов - в одной области действия может существовать множество одноимённых методов. При использовании форсирования компилятор должен быть способен устранить неоднозначности вызова методов и функций по числу или именам аргументов. Типы аргументов при устранении неоднозначностей не рассматриваются.

Абстрактные методы

Типовые методы объектов extern являются встроенными и реализуются целевой архитектурой. Программы P4 могут лишь вызывать такие методы.

Однако некоторые типы объектов extern могут предоставлять методы, которые можно реализовать в программе P4. Такие методы описываются с помощью ключевого слова abstract перед определением метода.

```
extern Balancer {
    Balancer();
    // Определение числа активных потоков
    bit<32> getFlowCount();
    // Возвращает индекс порта для балансировки нагрузки
    // @param address: Адрес IPv4 для источника потока
    abstract bit<4> on_new_flow(in bit<32> address);
}
```

При создании экземпляра такого объекта должна быть представлена реализация всех абстрактных методов (см. 10.3.1. Ограничения для создания экземпляров на верхнем уровне).

7.2.10. Специализация типа

Базовый тип может быть уточнён (специализирован) путём задания аргументов для переменных типа. В случаях, когда компилятор может вывести аргументы типа, такая специализация не требуется. Для специализированного типа все его переменные типа должны быть привязаны.

```
specializedType
        : prefixedType '<' typeArgumentList '>'
        ;
```

В приведённом ниже примере объявление extern описывает базовый блок регистров, где тип сохраняемых в регистре элементов является произвольным T.

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

Тип T должен указываться при создании экземпляра набора регистров путём задания типа Register

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

Создание экземпляра registerBank выполняется с использованием типа Register заданного привязкой bit<32> к аргументу типа T.

Типы struct, header, header_union и стек заголовков (header stack) также могут быть базовыми. Для использования таких базовых типов они должны быть заданы с использованием подобающих аргументов. Например,

```
// Базовый тип структуры
struct S<T> {
    T field;
    bool valid;
}

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

// Специализация S путём замены T на bit<32>
const S<bit<32>> s = { field = 32w0, valid = false };
// Специализация G путём замены T на bit<32>
const G<bit<32>> g = { s = { field = 0, valid = false } };

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

// Специализация H путём замены T на bit<8>
const H<bit<8>> h = { field = 1 };
```

```
// Стек заголовков создаётся из специализации базового типа header
H<bit<8>>[10] stack;

// Базовый тип header_union
header_union HU<T> {
    H<bit<32>> h32;
    H<bit<8>>  h8;
    H<T>       ht;
}

// Объединение заголовков с типом, получаемым специализацией базового типа header_union
HU<bit> hu;
```

7.2.11. Типы анализаторов и блоков управления

Типы parser и control похожи на тип function и описывают сигнатуру анализатора или блока управления. Такие функции не возвращают значений. Объявления анализаторов и блоков управления в архитектуре могут быть базовыми (т. е. иметь параметры типа).

Типы parser, control и package не могут служить типами аргументов для методов, анализаторов, элементов управления, таблиц или действий. Они могут служить типами для аргументов, передаваемых конструкторам (14. Параметризация).

7.2.11.1. Объявление типа анализатора

Объявление типа parser описывает сигнатуру синтаксического анализатора, которому следует иметь хотя бы один аргумент типа packet_in, представляющий принятый для обработки пакет.

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

Например, ниже показано объявление типа parser с именем P, параметризованное переменной типа H. Анализатор, получивший на входе packet_in со значением b, создаёт на выходе два значения:

- заданного пользователем типа H;
- предопределённого типа Counters.

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

7.2.11.2. Объявление типа блока управления

Объявление типа элемента управления (control) описывает сигнатуру блока управления и похоже на объявление типа parser.

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

7.2.12. Типы пакетов (package)

Тип package описывает сигнатуру программного пакета.

```
packageTypeDeclaration
        : optAnnotations PACKAGE name optTypeParameters
        '(' parameterList ')'
;
```

Все параметры package оцениваются при компиляции и не должны указывать направления (не могут быть in, out, inout). В остальном типы package похожи на parser. Для пакетов могут лишь создаваться экземпляры без управления поведением при работе.

7.2.13. Типы, не имеющие значения (don't care)

Не имеющие значения типы (_) могут применяться в некоторых обстоятельствах. Их следует использовать лишь там, где возможна запись переменной привязанного типа. Символ подчёркивания может применяться для снижения сложности кода, когда не имеет значения конкретная привязка переменной (например при унификации типа don't care может комбинироваться с любым другим типом). Пример рассмотрен в параграфе 16.1. Пример описания архитектуры.

7.3. Подразумеваемые значения

Для некоторых типов P4 определены «принятые по умолчанию значения», которые могут автоматически применяться при инициализации данного типа.

- Для int, bit<N> и int<N> подразумевается 0.
- Для bool подразумевается false.
- Для error подразумевается error.NoError (определено в core.p4).
- Для string подразумевается пустая строка "".
- Для varbit<N> подразумевается строка нулей (в P4 сейчас нет литерала для этого).
- Для enum с базовым типом подразумевается 0, даже если 0 реально не зазван среди значений enum.
- Для enum без базового типа подразумевается первое значение, появляющееся в объявлении типа enum.

- Для header подразумевается invalid.
- Для стека заголовков подразумевается, что все элементы являются invalid, a nextIndex = 0.
- Для header_union подразумевается invalid во всех элементах.
- Для struct подразумевается структура, где каждое поле имеет принятое по умолчанию значение в соответствии с типом поля, если такое значение определено.
- Для a tuple подразумевается кортеж, где каждое поле имеет принятое по умолчанию значение в соответствии с типом поля, если такое значение определено.

Отметим, что некоторые типы не имеют подразумеваемых значений, например, это match_kind, set, function, extern, parser, control, package.

7.4. typedef

Объявление typedef можно использовать для создания дополнительного имени типа.

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

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

Если два типа являются синонимами, все операции, которые могут быть выполнены над исходным типом, применимы и к вновь созданному.

7.5. Создание новых типов

Подобно typedef, можно использовать ключевое слово type для определения новых типов.

```
        | optAnnotations TYPE typeRef name
        | optAnnotations TYPE derivedTypeDeclaration name
```

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

В отличие от typedef, ключевое слово type создаёт новый тип, не являющийся синонимом существующего. Значения исходного и вновь созданного типа нельзя смешивать в выражениях.

Важным применением таких типов является описание значений P4, которыми нужно обмениваться с плоскостью управления по коммуникационным каналам (например, через API плоскости управления или в передаваемых этой плоскости сетевых пакетах). Архитектура P4 может, например, определять тип для портов коммутатора

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

Это определение предотвратит использование PortId_t в арифметических выражениях. Кроме того, такое определение может разрешить специальные манипуляции или такие значения программам, находящимся вне пути данных (например, специфические для платформы инструменты могут включать программы, автоматически преобразующие тип PortId_t в иное представление при обмена с программами плоскости управления).

8. Выражения

В этом разделе описаны все выражения, которые могут применяться в P4, с группировкой по типу выходных значений.

Правило грамматики для выражений общего назначение имеет вид

```
expression
        : INTEGER
        | TRUE
        | FALSE
        | STRING_LITERAL
        | nonTypeName
        | dotPrefix nonTypeName
        | expression '[' expression ']'
        | expression '[' expression ':' expression ']'
        | '{' expressionList '}'
        | '{' kvList '}'
        | '(' expression ')'
        | '!' expression
        | '~' expression
        | '-' expression
        | '+' expression
        | typeName '.' member
        | ERROR '.' member
        | expression '.' member
        | expression '*' expression
        | expression '/' expression
        | expression '%' expression
        | expression '+' expression
        | expression '-' expression
        | expression SHL expression          // SHL - это сдвиг влево (<<)
        | expression '>''>' expression       // символы >> не должны быть разделены
        | expression LE expression           // LE - это меньше или равно (<=)
        | expression GE expression
        | expression '<' expression
```

```
        | expression '>' expression
        | expression NE expression          // NE - не равно (!=)
        | expression EQ expression          // EQ - равно (==)
        | expression '&' expression
        | expression '^' expression
        | expression '|' expression
        | expression PP expression    // PP это ++
        | expression AND expression   // AND это &&
        | expression OR expression    // OR это ||
        | expression '?' expression ':' expression
        | expression '<' realTypeArgumentList '>' '(' argumentList ')'
        | expression '(' argumentList ')'
        | namedType '(' argumentList ')'
        | '(' typeRef ')' expression
        ;
expressionList
        : /* пусто */
        | expression
        | expressionList ',' expression
        ;
member
        : name
        ;
argumentList
        : /* пусто */
        | nonEmptyArgList
        ;
nonEmptyArgList
        : argument
        | nonEmptyArgList ',' argument
        ;
argument
        : expression
        ;
typeArg
        : DONTCARE
        | typeRef
        | nonTypeName
        ;
typeArgumentList
        : /* пусто */
        | typeArg
        | typeArgumentList ',' typeArg
        ;
```

Полная грамматика P4 приведена в Приложении H.

Приведённая грамматика не показывает очерёдности операций. Порядок действий похож на применяемый в C с одним исключением и некоторыми дополнениями. Приоритет побитовых операций &, | и ^ выше приоритета <, <=, >, >=. Это более естественно с учётом добавления логического типа true в систему типов, поскольку побитовые операции не могут применяться к логическим типам. Конкатенация (++) имеет такой же приоритет, что и инфиксное сложение. «нарезка» битов a[m:l] имеет такой же приоритет как индексирование массива (a[i]).

В дополнение к этому P4 поддерживает выражения выбора select (12.6. Выражения для выбора), которые могут применяться лишь в анализаторах.

8.1. Порядок операций

С учётом составных выражений, где оценивается порядок для субвыражений, важен учёт побочных эффектов в субвыражениях. Выражения в P4 оцениваются, как описано ниже.

- Логические операторы && и || оцениваются в сокращённом порядке, т. е. второй операнд учитывается лишь при необходимости.
- В условных операторах e1 ? e2 : e3 оценивается e1, затем e2 или e3.
- Все остальные выражения вычисляются слева направо в соответствии с порядком в коде программы.
- Вызовы методов и функций оцениваются в соответствии с параграфом 6.7. Соглашения о вызовах.

8.2. Операции над типом error

Символические имена, заданные определениями ошибок, относятся к пространству имён ошибок. Для типа error поддерживаются лишь операции сравнения «равно» (==) и «не равно» (!=), результатом которых является логическое значение. Например, приведённая ниже строка проверяет наличие определённой ошибки

```
error errorFromParser;

if (errorFromParser != error.NoError) { /* код опущен */ }
```

8.3. Операции над типом enum

Символические имена, заданные определениями перечисляемых типов, относятся к пространствам имён, созданным объявлениями enum, а не к пространству верхнего уровня.

```
enum X { v1, v2, v3 }
X.v1    // ссылка на v1
```

```
v1      // ошибка - v1 не относится к пространству имён верхнего уровня
```

Подобно error, выражения enum без указания базового типа поддерживают лишь операции сравнения равенства (==) и неравенства (!=). Выражения type без указания базового типа нельзя привести к иному типу (или из иного типа).

Для enum можно указать базовый тип, как показано ниже.

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

Можно отобразить несколько символьных значений enum на одно целое число с фиксированным значением.

```
enum bit<8> NonUnique {
        b1 = 0,
        b2 = 1, // b2 и b3 имеют одинаковые значения.
        b3 = 1,
        b4 = 2
}
```

Для enum с базовым типом поддерживается приведение базового типа, например

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

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

Значение a, которое было инициализировано как E.e2 приводится к типу bit<8> с использованием заданного представления целого числа без знака с фиксированным размером для E.e2 и значением 1. Затем для переменной b устанавливается символьное значение E.e2, которое соответствует целому числу без знака с фиксированным размером и значением 1.

Поскольку приведение enum к базовому целочисленному типу с фиксированным размером всегда безопасно, поддерживается также неявное приведение enum к целочисленному типу со знаком или без знака.

```
bit<8> x = E.e2; // устанавливает x = 1 (E.e2 автоматически приводится к bit<8>)

E a = E.e2
bit<8> y = a << 3; // устанавливает y = 8 (автоматически приводится к bit<8> и сдвигается)
```

Неявное приведение базового типа с фиксированным размером к enum не поддерживается.

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

enum bit<8> E2 {
        e1 = 10, e2 = 11, e3 = 12
}
E1 a = E1.e1;
E2 b = E2.e2;

a = b;  // Ошибка - b автоматически приводится к bit<8>,
        // но bit<8> не приводится автоматически к E1

a = (E1) b; // Корректно
a = E1.e1 + 1;          // Ошибка - E.e1 автоматически приводится к bit<8>
                        // и правая часть выражения имеет тип bit<8>,
                        // который нельзя автоматически привести к E.

a = (E1)(E1.e1 + 1); // Явное приведение типа делает назначение корректным

a = E1.e1 + E1.e2;      // Ошибка - оба аргумента сложения автоматически
                        // приводятся к bit<8>. Сложение корректно,
                        // но присваивание недействительно.

a = (E1)(E2.e1 + E2.e2); // Явное приведение делает присваивание корректным.
```

Благоразумный компилятор может выдавать предупреждение в случаях множественного автоматического приведения.

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

if (a > b) { // Возможно предупреждение о двух автоматический и разных приведениях к bit<8>.
               // Код опущен
}

c = a + b; // Корректно, но разумно предупредить.
```

Хотя приведение enum к целочисленному типу фиксированного размера без знака и обратно всегда безопасно, могут возникать случаи, когда приведение целого без знака к связанному с ним типу enum может давать значение без имени.

```
bit<8> x = 5;
E e = (E) x; // Устанавливает для e значение без имени.
```

Для e устанавливается неименованное значение, поскольку отсутствует символ, соответствующий целому числу без знака с фиксированным размером и значением 5. Например, в приведённом ниже фрагменте кода может быть

достигнута ветвь else в блоке if/else if/else, даже когда сопоставление с x выполнено относительно символов, определённых в MyPartialEnum_t.

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

bit<2> y = < некое значение >;
MyPartialEnum_t x = (MyPartialEnum_t)y;

if (x == MyPartialEnum_t.VALUE_A) {
        // Тот или иной код
} else if (x == MyPartialEnum_t.VALUE_B) {
        // Тот или иной код
} else if (x == MyPartialEnum_t.VALUE_C) {
        // Тот или иной код
} else {        // Компилятор P4 должен предполагать, что эта ветвь может быть выполнена
        // Тот или иной код
}
```

Кроме того, при использовании enum в качестве поля заголовка предполагается, что выбор перехода будет соответствовать принятому по умолчанию значению, если анализируемое целое значение не соответствует одному из символьных значений EtherType, как показано ниже.

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

header ethernet {
        // Часть полей опущена
        EtherType etherType;
}

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

Любая переменная типа enum, содержащая безымянное значение в результате приведения enum с базовым типом, синтаксического анализа в поле типа enum с базовым типом или просто объявления enum без начального значения, не будет совпадать с каким-либо из значений, заданных для этого типа. Такие безымянные значения все равно должны обеспечивать предсказуемое поведение в случаях, где они соответствуют какому-либо допустимому значению. Например, это должно соответствовать любой из приведённых ниже ситуаций.

- В выражениях select следует соответствовать варианту default или _ в выражении набора ключей.
- В качестве ключа с троичным match_kind в таблице следует соответствовать записи, где для всех битов поля установлено don't care.
- В качестве ключа с match_kind lpm следует соответствовать записи, где это поле имеет префикс размера 0.

Отметим, что при появлении enum без базового типа в API плоскости управления компилятор должен выбрать подходящий тип и представление данных сериализации. Для значений enum с базовым типом и представлением компилятору следует использовать заданный базовый тип в качестве данных сериализации и представления.

8.4. Логические выражения

Для логических выражений поддерживаются операции И (&&), ИЛИ (||), НЕ (!), равно (==) и не равно (!=). Порядок выполнения операций аналогичен принятому в C и использует сокращение (при очевидном результате расчёт прекращается). P4 не использует неявного приведения логических значений к bit-string и обратно. В результате обычные для C выражения вида

```
if (x) /* тело опущено */
```

(x - целочисленный тип) должно записываться в P4 как

```
if (x != 0) /* тело опущено */
```

В параграфе 8.9.2. Неявное приведение описана оценка 0 в этом выражении.

8.4.1. Условный оператор

Условные выражения вида e1 ? e2 : e3 ведут себя как в языке C. Как было отмечено выше, сначала оценивается выражение e1, затем e2 или e3 в зависимости от первого результата. Первое субвыражение должно иметь логический тип, а второе и третье должны быть однотипными, но не могут быть целочисленными с бесконечной разрядностью, если само условие не может быть оценено в момент компиляции. Это ограничение предназначено для того, чтобы размер результата условного выражения можно было статически вывести во время компиляции.

8.5. Операции над битовыми типами (целые числа без знака)

В этом разделе рассматриваются все операции, которые могут быть выполнены над выражениями типа bit<W> для заданного W (их называют также битовыми строками).

Арифметические операции «заворачиваются» подобно операциям C над целыми числами без знака (т. е. представление большого значения W битами сохраняет лишь младшие W битов этого значения). В частности, P4 не имеет арифметических исключений и результат арифметической операции определён для всех возможных входных значений.

Архитектура платформ P4 может поддерживать арифметику с насыщением. Все операции с насыщением ограничены фиксированным диапазоном от минимального до максимального значения. Арифметика с насыщением имеет преимущества, особенно при использовании в счётчиках. Результат достижения максимума в счётчике с насыщением значительно ближе к реальному результату, нежели переполнение счётчика и начало отсчёта с 0. Согласно Wikipedia, арифметика с насыщением численно близка к верному ответу, насколько это возможно. Для 8-битовой двоичной арифметики со знаком в случае корректного ответа 130 гораздо менее удивляет получение ответа 127 в арифметике с насыщением, нежели −126 от модульной арифметики. Аналогично для 8-битовой двоичной арифметики без знака при корректном ответе 258 меньшим сюрпризом будет получение ответ 255 от арифметики с насыщением, нежели 2 от модульной арифметики. В настоящее время P4 определяет операции с насыщением лишь для сложения и вычитания. Для целых чисел без знака размером W минимальным значением является 0, а максимальным 2^W-1. Приоритет операций сложения и вычитания с насыщением совпадает с приоритетом тех же операций в модульной арифметике.

Все двоичные операции (кроме сдвига) требуют двух операндов одного типа и размера, а предоставление операндов разного размера вызывает ошибку при компиляции. При оценке размера компилятор не применяет неявного приведения типов. Не поддерживается двоичных операций с комбинациями целых чисел со знаком и без знака (за исключением сдвига). Ниже перечислены операции над выражениями из битовых строк.

- Проверка равенства битовых строк одного размера обозначается == и даёт логический результат.
- Проверка неравенства битовых строк одного размера обозначается != и даёт логический результат.
- Сравнение беззнаковых чисел одного размера - <, >, <=, >= даёт логический результат.

Перечисленные ниже операции возвращают результат bit-string при работе с битовыми строками одного размера.

- Отрицание или смена знака (-) выполняется вычитанием значения из 2^W. Результатом служит целое число без знака с тем же размером. Семантика совпадает с принятой в C для целых чисел без знака.
- Унарный плюс (+) реально не делает ничего.
- Сложение (+) является ассоциативной операцией. Результат получается путём отсечки[1] полученной суммы до нужного размера (как в C).
- Вычитание (-). Результатом является целое число без знака с тем же типом, что и у операндов. Результат вычисляется путём сложения первого операнда с отрицанием второго (как в C).
- Умножение (*). Результат получается путём отсечки произведения до нужного размера. Архитектура P4 может вносить дополнительные ограничения, например, разрешать умножение лишь на степени 2.
- Побитовая операция И между двумя операндами одного размера (&).
- Побитовая операция ИЛИ между двумя операндами одного размера (|).
- Побитовое дополнение одной битовой строки (~).
- Побитовая операция Исключительное-ИЛИ (XOR) между двумя операндами одного размера (^).
- Сложение с насыщением (|+|).
- Вычитание с насыщением (|-|).

Для битовых строк поддерживаются также описанные ниже операции.

- Извлечение непрерывного набора битов, называемое «нарезкой» (slice), обозначается [m:l], где m и l должны быть положительными целыми числами, известными при компиляции и m >= l. Результатом является строка битов размером m - l + 1, включая биты от l (младший бит результата) до m (старший) исходного операнда. Выполнение условий 0 <= l < W и l <= m < W проверяется статически (W - размер исходной строки битов). Отметим, что обе конечные точки извлечения включаются в результат. Границы нужны в момент компиляции, чтобы можно было рассчитать размер результата. Нарезки относятся к l-value, поэтому P4 поддерживает для них присваивание e[m:l] = x . В этом случае устанавливаются биты от m до l строки e в соответствии с битами x, а остальные биты не меняются. «Нарезка» числа без знака является числом без знака.
- Логический сдвиг влево или вправо на известное в процессе работы целое число без знака обозначается << и >>. При сдвиге влево левый операнд является беззнаковым, а правый должен быть выражением типа bit<S> или неотрицательным целочисленным литералом. Результат имеет тот же тип, что был у левого операнда. Сдвиг на величину, превышающую размер, обнуляет все биты.

8.6. Операции над целыми числами фиксированного размера со знаком

В этом разделе рассмотрены все операции, которые могут быть выполнены над выражениями типа int<W> для данного W. Напомним, что int<W> обозначает целые числа со знаком W-bit, представленные дополнением до 2.

В общем случае арифметические операции P4 не детектируют переполнения или исчерпания - операции просто «начинают отсчёт с 0 (wrap around), как в операциях C с целыми числами без знака. Поэтому попытка представить большое число размером W будет сохранять лишь младшие W битов значения.

[1] *Старших битов. Прим. перев.*

P4 поддерживает арифметику с насыщением (сложение и вычитание) для целых чисел со знаком. Платформы могут отвергать арифметические операции с насыщением. Для целого числа со знаком размера W минимальным значением является $-2^{(W-1)}$, а максимальным $2^{(W-1)}-1$.

P4 не поддерживает арифметических исключений. Результат вычисления в процессе работы является определенным для всех комбинаций входных аргументов.

Все двоичные операции (кроме сдвига) требуют совпадения типа (наличие знака) и размера операндов. В противном случае компилятор сообщает об ошибке. Компилятор не использует неявного приведения для разнотипных операндов. Во всех операциях (кроме сдвига) P4 не позволяет комбинировать операнды со знаком и без знака.

Отметим, что побитовые операции для чисел со знаком определены чётко, поскольку такие числа всегда представляются в формате дополнения до 2.

Ниже перечислены операции, поддерживаемые для данных типа int<W>. Эти операции требуют однотипных операндов, а результат всегда имеет размер левого операнда.

- Отрицание или смена знака (-).
- Унарный плюс (+) реально не делает ничего.
- Сложение (+).
- Вычитание (-).
- Сравнение на равенство (==) и неравенство (!=), дающее логический результат.
- Численные сравнения <, <=, >, и >=, дающие логический результат.
- Умножение (*). Размер результата совпадает с размером операндов. Архитектура P4 может вносить дополнительные ограничения, например, умножение лишь на степени 2.
- Сложение с насыщением (|+|).
- Вычитание с насыщением (|-|).
- Арифметический сдвиг влево (<<) и вправо (>>). Левый операнд имеет знак, а правый должен быть беззнаковым типа bit<S> или неотрицательным целочисленным литералом. Результат всегда имеет тип левого операнда. Сдвиг влево ведёт себя точно так же, как для чисел без знака. Сдвиг на число, превышающее размер, ведёт к корректному результату:
  - все нули при сдвиге влево;
  - все нули при сдвиге вправо на положительное значение;
  - все 1 при сдвиге вправо на положительное значение.
- Извлечение непрерывного набора битов, называемое «нарезкой» (slice), обозначается [m:l], где m и l должны быть положительными целыми числами, известными при компиляции и m >= l. Результатом является целое число без знака размером m - l + 1, включая биты от l (младший бит результата) до m (старший) исходного операнда. Выполнение условий 0 <= l < W и l <= m < W проверяется статически (W - размер исходной строки битов). Отметим, что обе конечные точки извлечения включаются в результат. Границы нужны в момент компиляции, чтобы можно было рассчитать размер результата. Нарезки относятся к l-value, поэтому P4 поддерживает для них присваивание e[m:l] = x . В этом случае устанавливаются биты от m до l строки e в соответствии с битами x, а остальные биты не меняются. «Нарезка» числа без знака является числом без знака.

8.6.1. Конкатенация

Конкатенация применяется к паре битовых строк со знаком или без знака и обозначается инфиксным оператором ++. Результатом является строка битов, размер которой совпадает с суммой размеров операндов, а старшие биты берутся из левого операнда. Знак также определяется левым операндом.

8.6.2. Замечания о сдвиге

Сдвиги (со знаком или без знака) имеют ряд рассмотренных ниже особенностей.

- Сдвиг вправо отличается при для чисел со знаком и без него, сдвиг числа со знаком является арифметическим.
- Сдвиг на отрицательное значение не имеет явной семантики и система типизации P4 считает его некорректным.
- В отличие от C сдвиг на величину, превышающую размер даёт чётко определённый результат.
- В зависимости от возможностей платформы сдвиг может потребовать экспоненциальных операций по числу битов правого операнда.

Рассмотрим несколько примеров.

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

Как отмечено выше, P4 даёт строго определённый результат при сдвиге, превышающем размер (в отличие от C).

Платформы P4 могут вносить дополнительные ограничения для операций сдвига, такие как запрет сдвига по выражению, не являющемуся константой, или на величину, превышающую определённый предел.

8.7. Операции над целыми числами произвольной точности

Тип int обозначает целые числа произвольной точности (разрядности). Для всех выражений типа int значения должны быть известны в момент компиляции. Поддерживаемые для типа int операции P4 перечислены ниже.

- Отрицание или смена знака (-).
- Унарный плюс (+) реально не делает ничего.
- Сложение (+).
- Вычитание (-).
- Сравнение на равенство (==) и неравенство (!=), дающее логический результат.
- Численные сравнения <, <=, >, и >=, дающие логический результат.
- Умножение (*).
- Деление одного положительного значения на другое с отсечкой (/).
- Деление одного положительного значения на другое по модулю (%).
- Арифметический сдвиг влево (<<) и вправо (>>). Результат сдвига имеет тип int. Правый операнд должен быть положительной константой. Выражение a << b эквивалентно a × 2b, a a >> b - ⌊a/2b ⌋.

Все операнды приведённых выше действий должны иметь тип int. В двоичных операция не допускается комбинация значений типа int и значений типов с фиксированным размером. Однако в некоторых случаях компилятор автоматически выполняет приведение int к типам с фиксированным размером (8.9. Приведение типов).

Все расчёты со значениями int выполняются без потери информации. Например, перемножение двух 1024-битовых значений может дать результат размером 2048 битов (отметим, что конкретное представление значений int не задано). Значения int можно привести к типу bit<w> и int<w>. Приведение int к значению с фиксированные размером будет сохранять соответствующее число младших битов. При потере старших битов компилятору следует выдавать предупреждение.

Отметим, что побитовые операции (|, &, ^, ~) не определены для типа int. Кроме того, не допускаются операции деления (в том числе по модулю) на отрицательное число. Арифметика с насыщением не поддерживается для целых чисел с производной разрядностью.

8.8. Операции над битовыми строками переменного размера

Для поддержки анализа заголовков с полями переменного размера в P4 служит тип varbit. При каждом объявлении этого типа статически задаётся максимальный размер. До инициализации динамический размер битовой строки остаётся неизвестным. Для таких строк поддерживается ограниченный набор операций.

- Назначение другой строки битов переменного размера. Цель назначения должна иметь такой же статический размер, как источник. Во время присваивания значения для целевой переменной устанавливается динамический размер от источника.
- Сравнение двух однотипных полей varbit на равенство и неравенство. Поля varbit считаются равными при совпадении динамического размера и всех битов в рамках этого размера.

Перечисленные ниже операции не поддерживаются напрямую для типа varbit, но могут выполняться над другими типами, включающими поля varbit, для которых поддерживаются операции extract и emit (например, значения полей заголовков). Они упомянуты здесь лишь для облегчения поиска в разделе, посвящённом типу varbit.

- Анализатор извлекает данные в заголовок переменного размера с использованием двухаргументного метода extract во внешнем объекте packet_in (12.8.2. Извлечение при переменном размере), устанавливая динамический размер объекта.
- Метод emit внешнего объекта packet_out может выполняться для заголовков и некоторых других типов (15. Сборка пакета), содержащих поля varbit. Вызов emit вставляет битовую строку переменного размера с известным динамическим размером в создаваемый пакет.

8.9. Приведение типов

P4 поддерживает ограниченное приведение типов, записываемое в форме (t) e, где t указывает тип, а e - выражение. Приведение возможно лишь для базовых типов. Это может показаться неудобным для программистов, но обеспечивает ряд преимуществ:

- однозначное указание намерений пользователя;
- явное приведение числовых значений, которое может быть связано со значительными вычислительными издержками для значений со знаком на ряде платформ;
- снижается число ситуаций, описываемых в спецификации P4 (некоторые платформы могут не поддерживать приведение типов).

8.9.1. Явное приведение

Ниже перечислены корректные приведения типов в P4:

- bit<1> <-> bool преобразует 0 в false, 1 - в true и обратно;
- int<1> -> bool преобразует 0 в false, 1 - в true и обратно (иные значения не приводятся);
- int<W> -> bit<W> сохраняет все биты неизменными, переводя отрицательные числа в положительные;

- bit<W> -> int<W> сохраняет все биты неизменными, переводя числа со старшим битом 1 в отрицательные;
- bit<W> -> bit<X> отсекает значение при W > X и дополняет нулями при W <= X.
- int<W> -> int<X> отсекает значение при W > X и добавляет бит знака при W < X.
- int -> bit<W> преобразует целое число в достаточно большое дополнение до 2 для предотвращения потери информации, после чего отсекает результат до размера W; компилятору следует выдавать предупреждение о переполнении или преобразовании в отрицательное число;
- int -> int<W> преобразует целое число в достаточно большое дополнение до 2 для предотвращения потери информации, после чего отсекает результат до размера W; компилятору следует выдавать предупреждение о переполнении;
- приведение между парой типов, заданных typedef, эквивалентное одной из приведённых выше комбинаций;
- приведение между типом, заданным type, и исходным типом;
- приведение между enum с явным типом и исходным типом.

8.9.2. Неявное приведение

Для сохранения простоты языка и предотвращения скрытых приведений типа P4 неявно выполняет лишь приведение int к типам фиксированного размера и приведение enum к базовому типу. В частности, применение двоичной операции к выражению типа int и выражению с типом фиксированного размера приведёт к неявному приведению int к типу второго выражения. Например, для приведённых ниже объявлений

```
enum bit<8> E {
        a = 5;
}
bit<8>          x;
bit<16> y;
int<8>          z;
```

компилятор добавит неявные приведения, показанные ниже.

- x + 1 станет x + (bit<8>)1;
- z < 0 станет z < (int<8>)0;
- x << 13 станет 0 с выдачей предупреждения о переполнении;
- x | 0xFFF станет x | (bit<8>)0xFFF с выдачей предупреждения о переполнении;
- x + E.a станет x + (bit<8>)E.a.

8.9.3. Недействительные арифметические выражения

Многие из арифметических выражений, доступных в других языках, не разрешены в P4. Для иллюстрации рассмотрим пример.

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

В таблице показаны некоторые выражения, которые неприемлемы по причине нарушения правил типизации P4. Для каждого из таких выражений приводится вариант приемлемой записи. Отметим, что для некоторых выражений имеются несколько таких вариантов и они могут давать разные результаты. Компилятор не может угадать намерения пользователя, поэтому программист P4 должен сам выбрать нужный вариант.

Выражение	*Причина непригодности*	*Альтернативы*
x + y	Разные размеры	(bit<16>)x + y x + (bit<8>)y
x + z	Разные знаки	(int<8>)x + z x + (bit<8>)z
(int<8>)y	Невозможно изменить сразу знак и размер	(int<8>)(bit<8>)y (int<8>)(int<16>)y
y + z	Разные знаки и размеры	(int<8>)(bit<8>)y + z y + (bit<16>)(bit<8>)z (bit<8>)y + (bit<8>)z (int<16>)y + (int<16>)z
x << z	Левый операнд сдвига не может иметь знака	x << (bit<8>)z
x < z	Разные знаки	X < (bit<8>)z (int<8>)x < z
1 << x	Размер значения 1 неведом	32w1 << x
~1	Побитовая операция над int	~32w1
5 & -3	Побитовая операция над int	32w5 & -3

8.10. Операции над кортежами

Кортежи могут назначаться другим кортежам того же типа, передаваться как аргументы и возвращаться функциями, а также могут инициализироваться с выражениями списков.

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

Доступ к полям кортежа возможен с использованием синтаксиса индексов массива x[0], x[1]. Индексы должны быть константами во время компиляции, чтобы проверка типов могла статически идентифицировать поля.

В настоящее время поля кортежа не могут применяться в левой части выражений, даже если сам кортеж является левой частью. Т. е. значение можно задать (изменить) в выражении лишь для кортежа целиком, но не для отдельных его полей. Это ограничение может быть снято в будущих версиях языка.

8.11. Операции над списками

Выражения списков указываются в фигурных скобках с перечислением всех элементов через запятую.

```
expression ...
        | '{' expressionList '}'
expressionList
        : /* пусто */
        | expression
        | expressionList ',' expression
        ;
```

Типом выражения со списком является tuple (7.2.8. Синтезируемые типы данных). Выражения со списками могут присваиваться типам tuple, struct или header, а также передаваться методам в качестве аргументов, однако выражения со списками не являются l-value. Списки могут быть вложенными. Например, приведённый ниже фрагмент кода использует выражение со списком для одновременной передачи нескольких полей заголовка системе обучения.

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

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

Список можно применять для инициализации структуры, если число элементов списка совпадает с числом полей структуры. Результатом такой инициализации будет присвоение i-го элемента списка i-му полю структуры.

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

Выражения со списками можно также применять для инициализации переменных с типом tuple.

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

Пустой список имеет тип tuple<> - кортеж без элементов.

8.12. Выражения со значением struct

Можно написать выражение, оценка (расчёт) которого даёт тип struct или header. Синтаксис выражения имеет вид

```
expression ...
        | '{' kvList '}'
        | '(' typeRef ')' expression
        ;
kvList
        : kvPair
        | kvList "," kvPair
        ;
kvPair
        : name "=" expression
        ;
```

Для такого выражения typeRef задаёт имя типа структуры или заголовка и может быть опущено, если его можно вывести из контекста, например, при инициализации переменной с типом struct. Ниже приведён пример использования выражения в проверке равенства.

```
struct S {
        bit<32> a;
        bit<32> b;
}
S s;
// Сравнение с выражением типа struct
bool b = s == (S) { a = 1, b = 2 };
```

Выражения struct можно использовать в правой части операций присваивания, в сравнениях, выражениях полей выбора, а также в качестве аргументов функций, методов или действий. Эти выражения не могут быть l-value.

8.13. Операции над set

Некоторые выражения P4 обозначают наборы значений (set<T> для некого T, 7.2.8.1. Тип set). Эти выражения могут появляться лишь в ограниченном контексте - анализаторах и постоянных записях таблиц. Например, выражение select (12.6. Выражения для выбора) может иметь показанную ниже структуру.

```
select (expression) {
        set1: state1;
        set2: state2;
        // Остальные метки опущены
}
```

Здесь выражения set1, set2 и т. д. преобразуются в наборы значений, которые select проверяет на принадлежность к заданным меткам.

```
keysetExpression
        : tupleKeysetExpression
        | simpleKeysetExpression
        ;
```

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

simpleExpressionList
        : simpleKeysetExpression
        | simpleExpressionList ',' simpleKeysetExpression
        ;

simpleKeysetExpression
        : expression
        | DEFAULT
        | DONTCARE
        | expression MASK expression
        | expression RANGE expression
        ;
```

Операторы mask (&&&) и range (..) имеют одинаковый приоритет, который выше приоритета &.

8.13.1. Одноэлементные наборы

В контексте set (набор, множество) выражения обозначают одноэлементные наборы. Например, во фрагменте

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

метка 4 обозначает набор с единственным элементом 4.

8.13.2. Универсальный набор

В контексте set выражение default или _ обозначает универсальный набор, содержащий все возможные значения данного типа

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

8.13.3. Маски

Инфиксный оператор &&& принимает два аргумента типа bit<W> или сериализованных enum и создаёт значение типа set<ltype>, где ltype является типом левого аргумента. Значение справа служит маской, где каждый бит 0 указывает, что соответствующий бит не имеет значения (don't care). Более формально, набор, обозначенный a &&& b, определяется как

```
a &&& b = { c of type bit<W>, где a & b = c & b }
```

Например,

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

Обозначает набор из 16 разных 8-битовых значений с битовой маской XXXX1010, где X может принимать любое значение. Отметим, что может быть много способов указать набор ключей с использованием оператора маскирования, например, 8w0xFA &&& 8w0x0F даст такой же набор, что и приведённое выше выражение.

Архитектура P4 может вносить дополнительные ограничения на выражения в левой и правой части, например, требовать от одного или обоих быть известным при компиляции.

8.13.4. Диапазоны

Инфиксный оператор .. принимает два аргумента одного типа T, в качестве которого может служить bit<W> или int<W> и создаёт значение типа set<T>. Множество содержит все значения от левого до правого операнда включительно. Например,

```
4w5 .. 4w8
```

обозначает 4w5, 4w6, 4w7, 4w8.

8.13.5. Произведение множеств

Можно комбинировать множества (set) с помощью декартова произведения

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

Типом произведения множеств является множество кортежей.

8.14. Операции над типом struct

Единственной операцией для выражения типа struct является доступ к полям на основе нотации с точкой (.), например, s.field. Если s является l-value, то s.field также будет l-value. P4 позволяет копировать структуры с использованием операторов присваивания, когда источник и назначение одинаковы по структуре. Кроме того, структуру можно инициализировать выражением списка, как описано в параграфе 8.11. Операции над списками, или инициализатором структуры, как описано в следующем параграфе. В обоих случаях должны инициализироваться все поля структуры.

Две структуры можно сравнить на равенство (==) или неравенство (!=), если они имеют один тип и все поля могут быть сравнимы на равенство. Две структуры считаются равными тогда и только тогда, когда в них равны все соответствующие поля.

8.15. Инициализаторы структур

Структуры можно инициализировать с помощью выражений, значениями которых является структура (8.12. Выражения со значением struct), как показано ниже.

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

Компилятор должен выдавать ошибку, если имя поля появляется более одного раза в инициализаторе структуры. Описание поведения при чтении неинициализированных полей структуры дано в параграфе 8.22. Чтение неинициализированных значений и запись полей в недействительные заголовки.

8.16. Операции над заголовками

Для заголовков поддерживаются те же операции, что и для структур. Присвоение значения заголовку копирует также бит validity. Кроме того, для заголовков поддерживаются указанные ниже методы.

- isValid() возвращает значение бита validity в заголовке;
- setValid() устанавливает для бита validity значение true (применимо лишь к l-value);
- setInvalid() устанавливает для бита validity значение false (применимо лишь к l-value).

Выражение h.minSizeInBits() определено для любого h с типом header. Значением выражения является сумма размеров всех полей h, при этом поля varbit учитываются с размером 0. Выражение h.minSizeInBits() является константой в момент компиляции и имеет тип int. Выражение h.minSizeInBytes() похоже на h.minSizeInBits(), но возвращает общий размер полей заголовка в байтах с округлением в большую сторону, если размер заголовка не содержит целого числа байтов. h.minSizeInBytes() = (h.minSizeInBits() + 7) >> 3.

Подобно структурам, объект header можно инициализировать выражением со списком (8.11. Операции над списками), где содержатся все поля заголовка в нужном порядке или инициализатором структуры (8.14. Операции над типом struct). Инициализированный заголовок автоматически становится действительным.

```
header H { bit<32> x; bit<32> y; }
H h;
h = { 10, 12 };                // Это также делает заголовок h действительным
h = { y = 12, x = 10 };        // то же самое
```

Два однотипных заголовка можно сравнить на равенство (==) или неравенство (!=). Заголовки считаются равными при совпадении значений всех полей, включая validity. В параграфе 8.22. Чтение неинициализированных значений и запись полей в недействительные заголовки описано поведение в случаях считывания полей неинициализированного заголовка или записи в заголовок, который не является действительным.

8.17. Операции над стеком заголовков

Стек заголовков является массивом фиксированного размера с однотипными заголовками. Действительные элементы стека заголовков не обязаны быть непрерывными. P4 поддерживает набор операций для стеков заголовков. Стек заголовков hs типа h[n] можно описать приведённым ниже псевдокодом.

```
// объявление типа
bit hs_t {
        bit<32>         nextIndex;
        bit<32>         size;
        h[n]            data;   // обычный массив
}

// Объявление и создание экземпляра
hs_t hs;
hs.nextIndex = 0;
hs.size = n;
```

Интуитивно стек заголовков можно представить как структуру, содержащую обычный массив заголовков hs и счётчик nextIndex, который может служить для упрощения создания анализаторов для стеков заголовков, как описано ниже. Счётчик nextIndex инициализируется значением 0.

Для стека заголовков hs размером n приведённые ниже выражения будут действительны.

- hs[index] указывает заголовок в конкретной позиции внутри стека. Если hs является l-value, результат также будет l-value. Заголовок может быть недействительным. Некоторые реализации могут требовать, чтобы выражение index было известно при компиляции. Компилятор P4 должен выдавать ошибку, если значение индекса при компиляции выходит за рамки. Доступ к стеку заголовков hs по индексу меньше 0 или больше hs.size возвращает неопределённое значение (8.22. Чтение неинициализированных значений и запись полей в недействительные заголовки).

  Индекс является выражением, которое должно иметь один из указанных ниже типов:

  - int - целое число «бесконечной» точности (7.1.6.5. Целые числа “бесконечной точности”);
  - bit<W> - целое число без знака размером W битов (W >= 1, 7.1.6.2. Целые числа без знака (bit-string));
  - int<W> - целое число со знаком размером W битов (W >= 1, 7.1.6.3. Целые числа со знаком);
  - преобразованный в последовательную форму тип enum с базой bit<W> или int<W> (7.2.1. Перечисляемые типы).

- hs.size возвращает 32-битовое целое число без знака, указывающее размер стека заголовков (константа в момент компиляции).
- Присваивание из стека заголовков hs в другой стек требует, чтобы оба стека имели один тип и размер. Копируются все компоненты hs, включая его элементы, биты validity и nextIndex.

Чтобы помочь программистам создавать анализаторы для стеков заголовков, P4 также предлагает операции с автоматическим прохождением стека элементов при синтаксическом анализе.

- hs.next может применяться только в анализаторах и даёт ссылку на элемент стека с индексом hs.nextIndex. Если nextIndex не меньше размера стека, результатом оценки будет переход в состояние reject с ошибкой error.StackOutOfBounds. Если hs является l-value, индекс hs.next также будет l-value.
- hs.last может применяться только в анализаторах и даёт ссылку на элемент стека с индексом hs.nextIndex - 1, если такой элемент имеется. Если nextIndex меньше 1 или превышает размер, результатом оценки будет переход в состояние reject с установкой ошибки error.StackOutOfBounds. В отличие от hs.next ссылка никогда не является l-value.
- hs.lastIndex может применяться только в анализаторах и даёт 32-битовое целое число без знака, которое представляет индекс hs.nextIndex - 1. Если nextIndex = 0, оценка выражение ведёт к неопределённому состоянию.

P4 также предлагает несколько операций, которые можно использовать для манипуляций с вершиной и дном стека.

- hs.push_front(int count) смещает hs «вправо» на count и первые count элементов становятся недействительными, а последние count элементов стека отбрасываются. Счётчик hs.nextIndex увеличивается на count. Аргумент count должен быть положительным целым числом, известным при компиляции. Результат имеет тип void.
- hs.pop_front(int count) смещает hs «влево» на count (т. е. элемент count копируется в элемент 0). Последние count становятся недействительными, счётчик hs.nextIndex уменьшается на count. Аргумент count должен быть положительным целым числом, известным при компиляции. Результат имеет тип void.

Приведённый ниже псевдокод определяет поведение push_front и pop_front.

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

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

Два стека заголовков можно сравнить на равенство (==) или неравенство (!=) лишь в том случае, когда они имеют элементы одного типа и совпадают по размеру. Два стека считаются одинаковыми, если все их соответствующие элементы равны друг другу. Значение nextIndex не учитывается при проверке равенства.

8.18. Операции над объединениями заголовков

Переменная, объявленная с типом union, исходно недействительна, например

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

U u; // u недействительна
```

Это также предполагает, что каждый из заголовков h1 - hn в объединении заголовков исходно недействителен. В отличие от заголовков объединение невозможно инициализировать. Однако validity для объединения заголовков можно обновить путём присваивания действительного заголовка одному из элементов объединения.

```
U u;
H1 my_h1 = { 8w0 }; // my_h1 действителен
u.h1 = my_h1;          // u и u.h1 действительны
```

Можно также присвоить объединению список элементов

```
U u;
u.h2 = { 16w1 };       // u и u.h2 действительны
```

или установить биты validity напрямую

```
U u;
u.h1.setValid();       // u и u.h1 действительны
H1 my_h1 = u.h1;       // my_h1 действителен, но включает неопределённое значение
```

Отметим, что считывание неинициализированного заголовка даёт неопределённое значение, даже если заголовок действителен. Более формально - если u является выражением типа union U с полями диапазона hi, для манипуляций с u можно использовать указанные ниже операции.

- u.hi.setValid() устанавливает в битах validity значение true для заголовка hi и false для всех остальных, что означает возврат незаданного значения при чтении этих заголовков.
- u.hi.setInvalid() - если бит validity для любого члена u имеет значение true, для него устанавливается значение false, что означает возврат незаданного значения при чтении любого заголовка.

Можно считать присваивание объединению заголовков

```
u.hi = e
```

эквивалентом

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

если заголовок e действителен и

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

в противном случае.

Разрешены операции присваивания между однотипными объединениями заголовков. Присваивание u1 = u2 копирует полное состояние объединения u2 в u1. Если объединение u2 действительно, в нем имеется тот или иной действительный заголовок u2.hi. Присваивание будет вести себя как u1.hi = u2.hi. Если u2 не является действительным, u1 станет недействительным (т. е. все действительные заголовки в u1 станут недействительными).

Метод u.isValid() возвращает true, если действителен любой из элементов объединения и false - в остальных случаях. Методы setValid() и setInvalid() не определены для объединений заголовков.

Предоставление выражения типа union для выдачи (emit) ведёт к выдаче единственного действительного заголовка, если такой имеется. Ниже приведён пример использования объединения заголовков для унифицированного представления IPv4 и IPv6.

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

struct Parsed_packet {
        Ethernet ethernet;
        IP ip;
}

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

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

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

Можно также использовать объединения для анализа (выбранных) опций TCP, как показано ниже.

```
header Tcp_option_end_h {
        bit<8> kind;
}
header Tcp_option_nop_h {
        bit<8> kind;
}
header Tcp_option_ss_h {
        bit<8>  kind;
        bit<32> maxSegmentSize;
}
```

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

typedef Tcp_option_h[10] Tcp_option_stack;

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

parser Tcp_option_parser(packet_in b, out Tcp_option_stack vec) {
        state start {
                transition select(b.lookahead<bit<8>>()) {
                        8w0x0 : parse_tcp_option_end;
                        8w0x1 : parse_tcp_option_nop;
                        8w0x2 : parse_tcp_option_ss;
                        8w0x3 : parse_tcp_option_s;
                        8w0x5 : parse_tcp_option_sack;
                }
        }
        state parse_tcp_option_end {
                b.extract(vec.next.end);
                transition accept;
        }
        state parse_tcp_option_nop {
                b.extract(vec.next.nop);
                transition start;
        }
        state parse_tcp_option_ss {
                b.extract(vec.next.ss);
                transition start;
        }
        state parse_tcp_option_s {
                b.extract(vec.next.s);
                transition start;
        }
        state parse_tcp_option_sack {
                bit<8> n = b.lookahead<Tcp_option_sack_top>().length;
                // n указывает общий размер опции TCP SACK в байтах.
                // Размер varbit-поля sack в заголовке
                // Tcp_option_sack_h составляет, таким образом, n-2 байт.
                b.extract(vec.next.sack, (bit<32>) (8 * n - 16));
                transition start;
        }
}
```

Два однотипных объединения заголовков можно сравнить на равенство (==) и неравенство (!=). Объединения совпадают тогда и только тогда, когда в них совпадают все соответствующие поля (т. е., оба недействительны или в обоих соответствующие элементы действительны и значения действительных заголовков совпадают).

8.19. Вызовы методов и функций

Методы и функции можно вызывать с использованием показанного ниже синтаксиса.

```
expression
        : ...
        | expression '<' realTypeArgumentList '>' '(' argumentList ')'
        | expression '(' argumentList ')'
argumentList
        : /* пусто */
        | nonEmptyArgList
        ;
nonEmptyArgList
        : argument
        | nonEmptyArgList ',' argument
        ;
argument
        : expression   /* позиционный аргумент */
        | name '=' expression /* именованный аргумент */
        | DONTCARE
        ;
realTypeArgumentList
        : realTypeArg
```

```
        | realTypeArgumentList ',' typeArg
        ;
realTypeArg
        : DONTCARE
        | typeRef
        ;
```

При вызове функции или метода может дополнительно указываться имя параметра для каждого аргумента. Не допускается использованием имён лишь для части аргументов (все или ни одного). Аргументы функций оцениваются (вычисляются) в порядке их указания слева направо до вызова функции.

```
extern void f(in bit<32> x, out bit<16> y);
bit<32> xa = 0;
bit<16> ya;
f(xa, ya);      // соответствие аргументов по порядку
f(x = xa, y = ya);     // соответствие аргументов по именам
f(y = ya, x = xa);     // соответствие аргументов в произвольном порядке
//f(x = xa);   -- ошибка - недостаточно аргументов
//f(x = xa, x = ya); -- ошибка - аргумент указан дважды
//f(x = xa, ya); ошибка - некоторые аргументы заданы именами
//f(z = xa, w = yz); ошибка - нет параметров с именем z и w
//f(x = xa, y = 0);    ошибка - аргумент y должен быть l-value
```

При вызовах используется соглашение copy-in/copy-out (6.7. Соглашения о вызовах). Для базовых функций тип аргументов можно явно указывать при вызове функции. Компилятор вставляет неявное приведение для аргументов in в методах и функциях, отмеченных в разделе 8.9. Приведение типов. Типы всех остальных аргументов должны точно совпадать с типами параметров.

Возвращаемый функцией результат отбрасывается при вызове функции в качестве оператора.

Идентификатор _ (не имеет значения) можно использовать лишь в аргументах out, когда возвращаемое в аргументе значение игнорируется в последующих операциях. При использовании в базовых функциях и методах компилятор может отвергать программу, если он не способен вывести тип для аргумента don't care (_).

8.20. Вызовы конструкторов

Некоторые конструкции P4 обозначают ресурсы, выделяемые при компиляции. Это внешние объекты, анализаторы, блоки управления и пакеты (package). Выделение таких объектов может выполняться двумя способами:

- вызов конструктора, являющегося выражением, которое возвращает объект соответствующего типа;
- использование инициаторов, как описано в параграфе 10.3. Создание экземпляров.

Синтаксис вызова конструктора похож на вызов функции и может включать именованные аргументы. Конструкторы полностью оцениваются (вычисляются) при компиляции (17. Абстрактная машина P4 - оценка). Поэтому все аргументы конструктора должны быть выражениями, вычисляемыми во время компиляции.

Ниже приведён пример вызова конструктора для установки зависимого от платформы свойства таблицы.

```
extern ActionProfile {
        ActionProfile(bit<32> size);  // конструктор
}
table tbl {
        actions = { /* тело опущено */ }
        implementation = ActionProfile(1024);        // вызов конструктора
}
```

8.21. Операции над типами, заданными type

Для значений с типами, заданными ключевым словом type, поддерживается лишь несколько операций:

- присваивание l-value того же типа;
- сравнение на равенство или неравенство, если исходный тип поддерживает такое сравнение;
- приведение к исходному типу и обратно.

```
type bit<32> U32;
U32 x = (U32)0;                     // требуется приведение
U32 y = (U32) ((bit<32>)x + 1);     // приведение нужно для арифметики
bit<32> z = 1;
bool b0 = x == (U32)z;              // требуется приведение
bool b1 = (bit<32>)x == z;          // требуется приведение
bool b2 = x == y;                   // приведение не требуется
```

8.22. Чтение неинициализированных значений и запись полей в недействительные заголовки

Как отмечено в параграфе 8.17. Операции над стеком заголовков, любая ссылка на элемент стека заголовков hs[index], где index - известное во время компиляции выражение (константа), должно приводить к ошибке, если индекс выходит за пределы диапазона. В этом параграфе также определено поведение при работе выражений hs.next и hs.last и описанное здесь поведение имеет приоритет по отношению ко всем прочим описаниям этих выражений.

Все упоминания элементов стека заголовков в этом параграфе применяются лишь к выражениям hs[index], где индекс является переменной, определяемой в процессе работы, а не при компиляции. Реализации P4 могут не поддерживать hs[index], где индекс определяется переменной во время выполнения, но при поддержке таких выражений реализации следует соответствовать описанному здесь поведению.

В перечисленных ниже случаях считывание значения приведёт к использованию в поле незаданной величины.

- Чтение поля из недействительного (invalid) в данный момент заголовка.
- Чтение из действительного заголовка поля, не инициализированного с момента, когда заголовок стал действительным.
- Чтение любого другого неинициализированного заголовка (например, поля struct), неинициализированной переменной в действии или элементе управления или параметра за пределами вызванного элемента управления или действия, которому не было присвоено значение в процессе выполнения действия или элемента управления (этот список не является исчерпывающим).
- Чтение поля заголовка, который является элементом стека заголовков, где индекс выходит за пределы стека.

Вызов метода isValid() для элемента стека заголовков с индексом вне диапазона возвращает неопределённое логическое значение (true или false), но спецификация не требует определённого значения и даже согласованности значения при нескольких таких вызовах. Присваивание выходящего за пределы стека заголовков значения другому заголовку h ведёт к неопределённому состоянию h во всех переменных заголовка, включая бит validity.

Там, где упоминается заголовок, он может быть членом header_union, элементом стека заголовков или обычным заголовком. Такое неопределённое состояние может отличаться для разных операций считывания.

Для неинициализированного поля или переменной типа enum или error считывание незаданного значения может приводить к отличию от значений, определённых для этого типа. Такому неопределённому значению следует по-прежнему вести к предсказуемому поведению в случаях, когда будет подходить любое действительное значение, например, следует соответствовать в приведённых ниже случаях.

- При использовании в выражении select следует соответствовать default или _ в выражении набора ключей.
- При использовании в качестве ключа с троичным match_kind в таблице следует соответствовать записи, где все биты поля имеют флаг don't care.
- При использовании в качестве ключа match_kind lpm в таблице следует соответствовать записи, где поле имеет префикс размером 0.

Рассмотрим ситуацию, где header_union u1 включает u1.h1 и u1.h2, а в данной точке программы u1.h1 является действительным, u1.h2 - недействительным. При попытке записи в поле недействительного заголовка u1.h2 может измениться любое (или все) поле действительного заголовка u1.h1. Такая запись не должна менять бит validity какого-либо из членов u1 или другие состояния, определённые в данный момент в системе, независимо от места определения этих состояний. При выполнении любого из действий:

- запись в поле недействительного заголовка (обычного или элемента стека с индексом в пределах диапазона), не входящего в объединение заголовков (header_union);
- запись в элемент стека заголовков с индексом за пределами диапазона;
- вызов метода a setValid() или setInvalid() для элемента стека заголовков с индексом за пределами диапазона

не должно меняться какое-либо из определённых в системе состояний ни в полях заголовков, ни где-либо ещё. В частности, при вовлечении в запись недействительного заголовка он должен остаться недействительным. Любой записи в поля недействительного в данный момент заголовка или в элемент стека заголовков с индексом за пределами диапазона разрешено менять состояния с неопределёнными значениями (например, значения полей в недействительных заголовках).

Для анализатора или элемента управления верхнего уровня в архитектуре от этой архитектуры зависит задание параметров in или inout, которые будут инициализироваться при вызове элемента управления, условий, при которых будет выполняться инициализация, и значения для инициализации, если она выполняется.

Поскольку в P4 разрешены пустые кортежи и структуры, можно создать типы, значения которых не будут включать «полезной» информации. Например,

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

Ниже перечислены типы, которые считаются пустыми:

- строки битов размером 0;
- varbit размером 0;
- пустые кортежи (tuple<>);
- стеки размером 0;
- структуры без полей;
- кортежи, все поля которых относятся к пустым типам;
- структуры, все поля которых относятся к пустым типам.

Значения пустых типов не передают полезных сведений, в частности, их не нужно инициализировать явно для задания допустимых значений. В типах заголовков без полей всегда имеется бит validity.

8.23. Инициализация с принятыми по умолчанию значениями

Значения l-value можно инициализировать автоматически принятыми по умолчанию значениями подходящего типа с использованием синтаксиса … (7.3. Подразумеваемые значения). Значения типа struct, header, tuple можно инициализировать с использованием явно заданных и подразумеваемых значений, применяя нотацию ... в инициализаторе выражения списка (не заданные явно поля будут инициализированы принятыми по умолчанию

значениями). При инициализации struct, header и tuple с частичным заданием значений ... нужно размещать в конце инициализатора.

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

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

enum N1 {
        A, B, C, F
}

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

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

N0 n0 = ...;   // n0 инициализируется принятым по умолчанию значением 0
N1 n1 = ...;   // n1 инициализируется принятым по умолчанию значением N1.A
S s0 = ...;    // s0 инициализируется принятым по умолчанию значением { 0, false }
S s1 = { 1, ... };    // s1 инициализируется значением { 1, false }
S s2 = { b = true, ... }; // s2 инициализируется значением { 0, true }
T t0 = ...;    // t0 инициализируется значением { { 0, false }, 0, N1.A }
T t1 = { s = ..., ... }; // t1 инициализируется значением { { 0, false }, 0, N1.A }
T t2 = { s = ... }; // ошибка - нет инициализатора для полей n0 и n1
tuple<N0, N1> p = { ... }; // p инициализируется принятым по умолчанию значением { 0, N1.A }
T t3 = { ..., n0 = 2}; // ошибка - ... должно быть в конце
H h1 = ...;    // h1 инициализируется недействительным заголовком
H h2 = { f2=5, ... }; // h2 инициализируется действительным заголовком, f1 0, f2 5
H h3 = { ... };       // h3 инициализируется действительным заголовком, f1 0, f2 0
```

9. Объявление функции

Функции могут объявляться лишь на верхнем уровне и все параметры должны иметь направление. Язык P4 не позволяет определять рекурсивные функции.

```
functionDeclaration
        : functionPrototype blockStatement
        ;

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

Ниже приведён пример функции, возвращающей большее из двух 32-битовых значений.

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

Функции возвращают значение с помощью оператора return. Функция, возвращающая тип void может использовать return без аргумента, остальные функции должны возвращать значение подходящего типа при всех возможных вариантах исполнения.

10. Объявление констант и переменных

10.1. Константы

Для определения констант применяется показанный ниже синтаксис.

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

initializer
        : expression
        ;
```

Такое объявление вводит константу со значением указанного типа. Ниже приведено несколько корректных объявлений.

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

```
const Version version = { 32w0, 32w0 };
```

Выражение инициализатора должно быть известно в момент компиляции.

10.2. Переменные

Локальные переменные объявляются с типом и именем, а также могут указывать инициализатор и аннотацию.

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

optInitializer
        : /* пусто */
        | '=' initializer
        ;
```

Объявления переменных без инициализатора не инициализируются (за исключением заголовков и связанных с ними типов, которые инициализируются как недействительные (invalid) так же, как описано для параметров out в параграфе 6.7. Соглашения о вызовах). Язык вносит некоторые ограничения на типы переменных - можно применять большинство типов P4, которые могут быть записаны явно (например, базовые типы, struct, header, стеки заголовков, tuple). Однако невозможно объявить переменные типов, которые синтезируются компилятором (например, set). Кроме того, переменные типов parser, control, package, extern должны объявляться с инициализатором (10.3. Создание экземпляров). Считывание значения неинициализированной переменной даёт неопределённый результат. Компиляторам следует пытаться детектировать такие ситуации и выдавать предупреждения.

Ниже перечислены места в программах P4, где могут объявляться переменные.

- оператор block;
- состояние анализатора;
- тело действия (action);
- блок исполнения управляющего блока;
- список локальных объявлений анализатора;
- список локальных объявлений элемента управления.

Переменные имеют локальную область действия и ведут себя подобно созданным в стеке переменным C. Значение переменной никогда не сохраняется между вызовами включающего её блока. В частности, переменные не могут применяться для поддержки состояния между разными сетевыми пакетами.

10.3. Создание экземпляров

Создание экземпляров похоже на объявление переменных, но оно зарезервировано для типов с конструктором (внешние объекты, блоки управления, анализаторы и пакеты).

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

Создание экземпляра записывается как вызов конструктора с указанием имени и всегда выполняется во время компиляции (17.1. Известные при компиляции значения). Результатом служит создание объекта с указанным именем и его привязка к результату вызова конструктора. Отметим, что аргументы создания экземпляра можно задавать именами. Например, экземпляр гипотетического банка счётчиков можно создать, как показано ниже.

```
// Из библиотеки платформы
enum CounterType {
        Packets,
        Bytes,
        Both
}
extern Counter {
        Counter(bit<32> size, CounterType type);
        void increment(in bit<32> index);
}
// Программа пользователя
control c(/* параметры опущены */) {
        Counter(32w1024, CounterType.Both) ctr;      // создание экземпляра
        apply { /* тело опущено */ }
}
```

10.3.1. Ограничения для создания экземпляров на верхнем уровне

Программы P4 не могут создавать элементы управления и синтаксические анализаторы в пакете верхнего уровня (top-level package). Это ограничение нацелено на то, чтобы обеспечить размещение большинства состояний в самой архитектуре или локальном элементе управления синтаксического анализатора. Ниже приведён пример некорректного создания экземпляра.

```
// Программа
control c(/* параметры опущены */) { /* тело опущено */ }
c() c1; // Недопустимое создание экземпляра на верхнем уровне
```

Элемент управления c1 создаётся на верхнем уровне, что недопустимо. Отметим, что объявление на верхнем уровне констант и создание экземпляров внешних объектов являются корректными действиями.

11. Операторы

Каждый оператор P4 (кроме block) должен завершаться точкой с запятой (;). Операторы могут указываться внутри:

- состояний анализатора;
- блоков управления;
- действий.

Для размещения операторов в конкретных местах имеются ограничения. Например, в анализаторе не поддерживаются условные операторы, а операторы switch допускаются только в блоках управления. Ниже представлен общий случай для блока управления.

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

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

Кроме того, анализаторам следует поддерживать оператор transition (12.5. Операторы смены (перехода) состояния).

11.1. Оператор присваивания

В операторах присваивания (=) сначала проверяется, относится ли левое выражение к l-value, затем определяется значение правого выражения и результат копируется в l-value. Производные типы (например, struct) копируются рекурсивно с копированием всех полей заголовка, включая биты validity. Присваивание не определено для внешних объектов.

11.2. Пустой оператор

Пустой оператор имеет вид ; (нет операций).

```
emptyStatement
        : ';'
        ;
```

11.3. Оператор блока

Оператор блока обозначается фигурными скобками и содержит последовательность операторов и определений, выполняемых по порядку. Переменные, константы и экземпляры объектов, созданные в блоке, видны только внутри этого блока.

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

statOrDeclList
        : /* пусто */
        | statOrDeclList statementOrDeclaration
        ;

statementOrDeclaration
        : variableDeclaration
        | constantDeclaration
        | statement
        | instantiation
        ;
```

11.4. Оператор возврата

Оператор return незамедлительно прерывает выполнение действия, функции или блока управления. Оператор return не разрешается использовать в анализаторах. В функциях с возвращаемым значением за оператором return следует выражение, тип которого должен соответствовать объявленному типу функции. Любые действия в стиле copy-out, заданные параметрами out или inout во включающем оператор действии, функции или элементе управления, происходят после выполнения оператора return (6.7. Соглашения о вызовах).

```
returnStatement
        : RETURN ';'
        | RETURN expression ';'
        ;
```

11.5. Оператор выхода

Оператор exit незамедлительно прерывает выполнение всех выполняющихся в данное время блоков. Прерывается текущее действие (если оператор вызван внутри action), текущий элемент управления и все вызывающие их. Оператор exit нельзя использовать в анализаторах и функциях.

Любые действия copy-out, заданные параметрами с направлением out или inout включающего оператор действия или элемента управления и всех вызывающих его, выполняются после выполнения оператора exit (6.7. Соглашения о вызовах.

```
exitStatement
        : EXIT ';'
        ;
```

11.6. Условный оператор

В условных операторах используется стандартный синтаксис и семантика, принятые в других языках программирования. Однако условные выражения в P4 должны быть логическими (а не целочисленными). Условные операторы нельзя применять в синтаксических анализаторах.

```
conditionalStatement
        : IF '(' expression ')' statement
        | IF '(' expression ')' statement ELSE statement
        ;
```

Во вложенных конструкциях с условиями else относится к внутреннему (наиболее близкому) оператору if, у которого нет оператора else.

11.7. Оператор выбора

Оператор switch можно применять только в блоках управления.

```
switchStatement
        : SWITCH '(' expression ')' '{' switchCases '}'
        ;

switchCases
        : /* пусто */
        | switchCases switchCase
        ;

switchCase
        : switchLabel ':' blockStatement
        | switchLabel ':'     // «пропуск» (fall-through)
        ;

switchLabel
        : name
        | DEFAULT
        ;

nonBraceExpression
        : INTEGER
        | TRUE
        | FALSE
        | STRING_LITERAL
        | nonTypeName
        | dotPrefix nonTypeName
        | nonBraceExpression '[' expression ']'
        | nonBraceExpression '[' expression ':' expression ']'
        | '(' expression ')'
        | '!' expression %prec PREFIX
        | '~' expression %prec PREFIX
        | '-' expression %prec PREFIX
        | '+' expression %prec PREFIX
        | typeName '.' member
        | ERROR '.' member
        | nonBraceExpression '.' member
        | nonBraceExpression '*' expression
        | nonBraceExpression '/' expression
        | nonBraceExpression '%' expression
        | nonBraceExpression '+' expression
        | nonBraceExpression '-' expression
        | nonBraceExpression '|+|' expression
        | nonBraceExpression '|-|' expression
        | nonBraceExpression '<<' expression
        | nonBraceExpression '>>' expression
        | nonBraceExpression '<=' expression
        | nonBraceExpression '>=' expression
        | nonBraceExpression '<' expression
        | nonBraceExpression '>' expression
        | nonBraceExpression '!=' expression
        | nonBraceExpression '==' expression
        | nonBraceExpression '&' expression
        | nonBraceExpression '^' expression
        | nonBraceExpression '|' expression
        | nonBraceExpression '++' expression
        | nonBraceExpression '&&' expression
        | nonBraceExpression '||' expression
        | nonBraceExpression '?' expression ':' expression
        | nonBraceExpression '<' realTypeArgumentList '>' '(' argumentList ')'
        | nonBraceExpression '(' argumentList ')'
        | namedType '(' argumentList ')'
        | '(' typeRef ')' expression
```

```
    ;
```

Выражение без скобок nonBraceExpression похоже на expression (раздел 8. Выражения), но не включает вариантов, начинающихся с левой (открывающей) скобки { для предотвращения синтаксической неоднозначности в операторах блоков.

Разрешены два типа выражений switch, описанных в двух следующих параграфах.

11.7.1 Оператор switch с выражением action_run

Выражение в операторе switch должно быть результатом вызова таблицы (13.2.2. Вызов блока СД).

Если за меткой оператора switch нет оператора блока, выполняется переход к следующей метке. Однако при наличии оператора блока переход к следующей метке не происходит. Это отличается от операторов выбора в C, где для предотвращения перехода к следующей метке требуется break. Допускается отсутствие действий для некоторых меток и отсутствие метки default. Если в процессе работы не найдено совпадений, просто продолжается выполнение программы. Повторение метки в операторе switch не допускается.

```
switch (t.apply().action_run) {
        action1:        // переход к action2:
        action2: { /* тело опущено */ }
        action3: { /* тело опущено */ }       // нет перехода от action2 к action3
        default: { /* тело опущено */ }
}
```

Отметим, что метка default в операторе switch используется для выполнения того или иного действия, независимо от нахождения или отсутствия ключа в таблице. Метка default не указывает отсутствие таблицы и заданное по умолчанию действие (default_action) выполняется. Метка default (при наличии) должна быть последней.

11.7.2 Оператор switch с выражением целочисленного или перечисляемого типа

Для этого варианта оператора switch выражение должно давать в результате один из указанных ниже типов:

- bit<W>;
- int<W>;
- enum с указанным базовым представлением или без него;
- error.

Все метки switch должны быть выражениями, значения которых известны при компиляции, а также должны иметь тип, который может неявно приводиться к типу выражения switch (8.9.2. Неявное приведение). Меткам switch недопустимо начинаться с левой фигурной скобки { во избежание неоднозначности восприятия блоков.

```
// Предположим, что выражение hdr.ethernet.etherType имеет тип bit<16>
switch (hdr.ethernet.etherType) {
   0x86dd: { /* тело опущено */ }
   0x0800:           // пропуск с переходом к следующему телу.
   0x0802: { /* тело опущено */ }
   0xcafe: { /* тело опущено */ }
   default: { /* тело опущено */ }
}
```

11.7.3 Замечания для всех операторов switch

Если две метки оператора switch совпадают, возникает ошибка при компиляции. Значениям меток switch не обязательно включать все возможные значения выражения switch и можно иметь принятый по умолчанию вариант (метку) switch, который при его наличии должен быть последним в операторе.

Если за меткой switch не следует оператор блока, выполняется переход к следующей метке. Однако при наличии оператора блока такого перехода не происходит. Отметим, что это отличается от операторов switch в языке C, где для предотвращения перехода в следующей метке требуется break. Если за последней меткой switch нет оператора блока, это обрабатывается как наличие пустого блока { }.

При выполнении оператора switch сначала вычисляется выражение switch и все побочные эффекты такого вычисления видимы для всех исполняемых вариантов switch. Из числа меток, не заданных по умолчанию, может быть не более одной, совпадающей со значением выражения switch. Если такая метка имеется, выполняется соответствующий вариант switch.

Если ни одна из меток не совпадает со значением выражения switch:

- при наличии метки default выполняется заданный по умолчанию вариант;
- если метка default не задана, не выполняется ни один из вариантов switch и исполнение продолжается после завершения оператора switch без побочных эффектов (если их нет при вычислении выражения switch).

Обработка обобщённых операторов описана в отдельном файле.

12. Анализ пакета

В этом разделе описаны конструкции P4 относящиеся к синтаксическому анализу пакетов.

12.1. Состояния анализатора

Синтаксический анализатор P4 описывает конечный автомат с одним стартовым и двумя финальными состояниями. Начальное состояние всегда именуется start. Два финальных состояния называются accept (успешный анализ) и reject (отказ при анализе). Состояние start является частью анализатора, а состояния accept и reject отличаются от

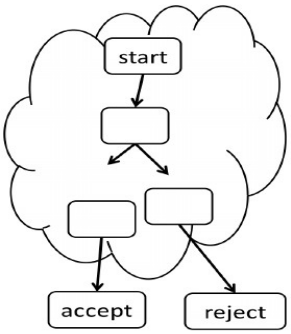

Рисунок 8. Структура FSM.

состояний, задаваемых программистом, и логически находятся за пределами анализатора. На рисунке 8 показана общая структура конечного автомата анализатора (FSM).

12.2. Объявление анализатора

Объявление синтаксического анализатора включает имя, список параметров, необязательный список параметров конструктора, локальные элементы и состояния анализатора (а также дополнительные аннотации).

```
parserTypeDeclaration
        : optAnnotations PARSER name optTypeParameters
          '(' parameterList ')'
        ;
parserDeclaration
        : parserTypeDeclaration optConstructorParameters
          '{' parserLocalElements parserStates '}'
        ;
parserLocalElements
        : /* пусто */
        | parserLocalElements parserLocalElement
        ;
parserStates
        : parserState
        | parserStates parserState
        ;
```

Описание optConstructorParameters, полезных при создании параметризованных анализаторов, приведено в разделе 14. Параметризация.

В отличие от объявления типа анализатора, объявление самого анализатора не может быть базовым. Например, приведённое ниже определение недействительно.

```
parser P<H>(inout H data) { /* тело опущено */ }
```

Поэтому при использовании в контексте parserDeclaration правило создания parserTypeDeclaration не должно давать параметры типа.

В любом анализаторе должно присутствовать по меньшей мере одно состояние - start. Не разрешается определять несколько состояний с одним именем, а также не допускается явное создание в анализаторе состояний с именами accept и reject - эти состояния логически отличаются от определяемых программистом.

Объявления состояний описаны ниже. Перед состояниями анализатор может также указывать локальные элементы, которыми могут быть константы и переменные, а также создание экземпляров объектов, которые могут применяться в анализаторе. Это могут быть экземпляры внешних объектов или другие анализаторы, вызываемые как подпрограммы. Однако не допускается создание в анализаторе блоков управления.

```
parserLocalElement
        : constantDeclaration
        | variableDeclaration
        | valueSetDeclaration
        | instantiation
        ;
```

Пример с полным объявлением анализатора приведён в параграфе 5.3. Полная программа VSS.

12.3. Абстрактная машина синтаксического анализа

Семантику синтаксического анализатора P4 можно выразить через абстрактную машину, манипулирующую структурой данных ParserModel. Ниже эта машина описана псевдокодом. Выполнение анализатора начинается из состояния start и завершается по достижении состояния reject или accept.

```
ParserModel {
        error  parseError;
        onPacketArrival(packet p) {
                ParserModel.parseError = error.NoError;
                goto start;
        }
}
```

Архитектура должна задавать поведение при достижении состояний accept и reject. Например, можно задать отбрасывание пакетов по достижении состояния reject без дальнейшей обработки. Другим вариантом может быть передача таких пакетов следующему за анализатором блоку с внутренними метаданными, показывающими состояние reject и причину его.

12.4. Состояния анализатора

Состояния анализатора объявляются в виде

```
parserState
        : optAnnotations STATE name
        '{' parserStatements transitionStatement '}'
        ;
```

Каждое состояние имеет имя и тело, состоящее из последовательности операторов, описывающих обработку, выполняемую при переходе анализатора в данное состояние, включая:

- объявления локальных переменных;
- операторы присваивания;
- вызовы методов, включая:
  - исполнение функций (например, verify для проверки действительности уже проанализированных данных);
  - исполнение методов (например, извлечение данных из пакета или расчёт контрольной суммы) и других анализаторов (12.10. Субанализаторы);
- условные операторы;
- переходы в другое состояние (12.5. Операторы смены (перехода) состояния).

Синтаксис операторов для анализатора использует приведённые ниже правила.

```
parserStatements
        : /* пусто */
        | parserStatements parserStatement
        ;
parserStatement
        : assignmentOrMethodCallStatement
        | directApplication
        | variableDeclaration
        | constantDeclaration
        | parserBlockStatement
        | emptyStatement
        | conditionalStatement
        ;
parserBlockStatement
        : optAnnotations '{' parserStatements '}'
        ;
```

Архитектура может вносить ограничения для выражений и операторов в анализаторах. Например, может запрещаться использование таких операций, как умножение или ограничиваться число используемых локальных переменных.

В ParserModel операторы в том или ином состоянии выполняются последовательно.

12.5. Операторы смены (перехода) состояния

Последним оператором в состоянии анализатора является необязательный оператор transition, переводящий элемент управления в иное состояние (возможно, accept или reject). Синтаксис оператора transition показан ниже.

```
transitionStatement
        : /* пусто */
        | TRANSITION stateExpression
        ;

stateExpression
        : name ';'
        | selectExpression
        ;
```

Выполнение оператора transition вызывает оценку (вычисление) stateExpression и переход в соответствующее состояние. В ParserModel семантику перехода можно формализовать в виде

```
goto eval(stateExpression)
```

Например, оператор

```
transition accept;
```

прерывает работу текущего анализатора незамедлительным переходом в состояние accept.

Если тело блока состояний не включает оператора transition, подразумевается переход в состояние reject.

12.6. Выражения для выбора

Выражение select ведёт к выбору состояния и использует показанный ниже синтаксис.

```
selectExpression
        : SELECT '(' expressionList ')' '{' selectCaseList '}'
        ;

selectCaseList
        : /* пусто */
        | selectCaseList selectCase
        ;

selectCase
        : keysetExpression ':' name ';'
```

```
        ;

selectCase
        : keysetExpression ':' name ';'
        ;
```

Если в выражении select expressionList имеет тип tuple<T>, каждое из выражений keysetExpression должно иметь тип set<tuple<T>>.

В ParserModel выражение select

```
select(e) {
        ks[0]: s[0];
        ks[1]: s[1];
        /* остальные варианты опущены */
        ks[n-2]: s[n-1];
        _ : sd; // ks[n-1] используется по умолчанию
}
```

можно представить в виде псевдокода

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

Некоторые платформы могут требовать, чтобы все выражения keyset в select были известны в момент компиляции. Выражения keyset оцениваются по порядку сверху вниз, как показано в приведённом выше псевдокоде. Первое из значений keyset, включающее метку select, обеспечит выбор результирующего состояния. Если соответствующей метки не найдено, при выполнении возникает ошибка со стандартным кодом error.NoMatch.

Приведённый пример предполагает, что все метки после default или _ не используются. Компилятор в таких случаях должен выдавать предупреждение. Это показывает важное различие между выражениями select и операторами switch в других языках, поскольку keyset в выражениях select могут «перекрываться».

Типичным примером использования select является сравнение недавно извлечённого поля заголовка с набором констант, как показано ниже

```
header IPv4_h { bit<8> protocol; /* другие поля опущены */ }
struct P { IPv4_h ipv4; /* другие поля опущены */ }
P headers;
select (headers.ipv4.protocol) {
        8w6     : parse_tcp;
        8w17    : parse_udp;
        _       : accept;
}
select (p.tcp.port) {
        16w0 &&& 16w0xFC00: well_known_port;
        _: other_port;
}
```

Выражение 16w0 &&& 16w0xFC00 описывает набор 16-битовых значений и 0 в шести старших битах.

Некоторые платформы могут поддерживать наборы значений в анализаторах (12.11. Набор значений анализатора). Для параметра типа T в наборе значений типом набора будет set<T>. Тип набора значений должен совпадать с типом всех других keysetExpression в данном выражении select и при наличии расхождений компилятор должен выдавать ошибку. В наборе значений должен использоваться тип bit<>, tuple или struct.

Например, чтобы позволить API плоскости управления задавать порты TCP в процессе работы, можно использовать

```
struct vsk_t {
        @match(ternary)
        bit<16> port;
}
value_set<vsk_t>(4) pvs;
select (p.tcp.port) {
        pvs: runtime_defined_port;
        _: other_port;
}
```

Приведённый пример позволяет API в процессе работы заполнить до 4 разных keysetExpression в value_set. Если value_set принимает в качестве параметра типа struct, API может использовать имена полей struct для указания объектов в наборе значений. Тип совпадения для поля struct задаётся аннотацией @match. Если эта аннотация не задана для поля struct, предполагается @match(exact). Одиночное поле без точного совпадения должно помещаться в struct самим полем, желательно с аннотацией @match.

12.7. Оператор verify

Оператор verify обеспечивает простую форму обработки ошибок и может вызываться лишь из анализаторов. Синтаксически использование оператора похоже на вызов функции

```
extern void verify(in bool condition, in error err);
```

Если первый аргумент имеет значение true, выполнение оператора не имеет побочных эффектов. Однако при значении false оператор вызывает незамедлительный переход в состояние reject с прерыванием анализа и установкой в parserError, связанного с анализатором, значения второго аргумента. В ParserModel семантика verify имеет вид

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

```
                goto reject;
        }
}
```

12.8. Извлечение данных

Основная библиотека P4 включает объявление встроенного extern типа packet_in, представляющего входные пакеты из сети. Этот тип является особым и пользователь не может явно создать его экземпляр. Архитектура представляет отдельный экземпляр packet_in при создании каждого экземпляра анализатора.

```
extern packet_in {
        void extract<T>(out T headerLvalue);
        void extract<T>(out T variableSizeHeader, in bit<32> varFieldSizeBits);
        T lookahead<T>();
        bit<32> length();      // Метод поддерживается не всеми архитектурами
        void advance(bit<32> bits);
}
```

Для извлечения данных из пакета, представленного аргументом b типа packet_in, анализатор вызывает методы извлечения для b. Имеется два варианта метода извлечения - с одним аргументом для заголовков фиксированного размера и с двумя аргументами для заголовков переменного размера. Поскольку эти операции могут приводить к отказам проверок во время работы, выполнение их возможно лишь из синтаксических анализаторов.

При извлечении данных в строку битов или целое число первый бит пакета помещается в старший бит.

Некоторые платформы могут выполнять обработку «на лету» (cut-through), т. е. начинать обработку до того, как станет известен размер пакета (пакет ещё не принят полностью). На таких платформах вызов метода packet_in.length() не может быть реализован и попытки таких вызовов должны помечаться как ошибки (во время компиляции back-end или при попытке загрузки скомпилированной программы P4 на платформе, не поддерживающей этот метод).

В ParserModel семантику packet_in можно представить с использованием абстрактной модели пакетов

```
packet_in {
        unsigned nextBitIndex;
        byte[] data;
        unsigned lengthInBits;
        void initialize(byte[] data) {
                this.data = data;
                this.nextBitIndex = 0;
                this.lengthInBits = data.sizeInBytes * 8;
        }
        bit<32> length() { return this.lengthInBits / 8; }
}
```

12.8.1. Извлечение при фиксированном размере

Метод извлечения с одним аргументом, применяемый для заголовков фиксированного размера, объявлен в форме

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

Выражение headerLeftValue должно оцениваться в l-value (6.6. Выражения для левой части) типа header с фиксированным размером. При успешном выполнении метода headerLvalue заполняется данными из пакета и для бита validity устанавливается значение true. Метод может приводить к отказам, например, при нехватке в пакете битов для заполнения заголовка. Ниже приведён фрагмент программы для извлечения заголовка Ethernet.

```
struct Result { Ethernet_h ethernet; /* другие поля опущены */ }
parser P(packet_in b, out Result r) {
        state start {
                b.extract(r.ethernet);
        }
}
```

В ParserModel семантику извлечения с одним аргументом можно представить приведённым ниже псевдокодом. Специальный идентификатор valid$ служит для скрытого флага validity, isNext$ - для индикации получения l-value с использованием next, а nextIndex$ - для указания соответствующих свойств стека заголовков.

```
void packet_in.extract<T>(out T headerLValue) {
        bitsToExtract = sizeofInBits(headerLValue);
        lastBitNeeded = this.nextBitIndex + bitsToExtract;
        ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
        headerLValue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
        headerLValue.valid$ = true;
        if headerLValue.isNext$ {
                verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
                headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
        }
        this.nextBitIndex += bitsToExtract;
}
```

12.8.2. Извлечение при переменном размере

Извлечение с двумя аргументами для заголовков переменного размера объявляется в P4 в форме

```
void extract<T>(out T headerLvalue, in bit<32> variableFieldSize);
```

Выражение headerLvalue должно быть l-value и представлять заголовок с единственным полем varbit. Выражение variableFieldSize должно оцениваться в значение bit<32>, указывающее число битов, извлекаемых в уникальное поле varbit переменной header (т. е. это не размер заголовка, а размер поля varbit). В ParserModel семантику извлечения с двумя аргументами можно выразить псевдокодом, приведённым ниже.

```
void packet_in.extract<T>(out T headerLvalue,
                                        in bit<32> variableFieldSize) {
        // платформы могут, но не обязаны включать строку проверки
```

```
        // verify(variableFieldSize[2:0] == 0, error.ParserInvalidArgument);
        bitsToExtract = sizeOfFixedPart(headerLvalue) + variableFieldSize;
        lastBitNeeded = this.nextBitIndex + bitsToExtract;
        ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
        ParserModel.verify(bitsToExtract <= headerLvalue.maxSize, error.HeaderTooShort);
        headerLvalue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
        headerLvalue.varbitField.size = variableFieldSize;
        headerLvalue.valid$ = true;
        if headerLValue.isNext$ {
                verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
                headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
        }
        this.nextBitIndex += bitsToExtract;
}
```

Приведённый ниже пример показывает один из способов анализа опций IPv4 путём разделения заголовка IPv4 на два отдельных объекта header.

```
// Заголовок IPv4 без опций
header IPv4_no_options_h {
        bit<4>          version;
        bit<4>          ihl;
        bit<8>          diffserv;
        bit<16> totalLen;
        bit<16> identification;
        bit<3>          flags;
        bit<13> fragOffset;
        bit<8>          ttl;
        bit<8>          protocol;
        bit<16> hdrChecksum;
        bit<32> srcAddr;
        bit<32> dstAddr;
}
header IPv4_options_h {
        varbit<320> options;
}
struct Parsed_headers {
        // некоторые поля опущены
        IPv4_no_options_h ipv4;
        IPv4_options_h ipv4options;
}

error { InvalidIPv4Header }

parser Top(packet_in b, out Parsed_headers headers) {
        // некоторые состояния опущены
        state parse_ipv4 {
                b.extract(headers.ipv4);
                verify(headers.ipv4.ihl >= 5, error.InvalidIPv4Header);
                transition select (headers.ipv4.ihl) {
                        5: dispatch_on_protocol;
                        _: parse_ipv4_options;
                }
        }
        state parse_ipv4_options {
                // используется информация из заголовка IPv4 для расчёта
                // числа извлекаемых битов
                b.extract(headers.ipv4options,
                                (bit<32>)(((bit<16>)headers.ipv4.ihl - 5) * 32));
                transition dispatch_on_protocol;
        }
}
```

12.8.3. Предварительный просмотр

Метод lookahead, обеспечиваемый абстракцией packet_in, оценивает набор битов из входящего пакета без указателя nextBitIndex. Подобно extract, он будет переводить в состояние reject и возвращать ошибку, если в пакете недостаточно битов. Вызов метода имеет вид

```
b.lookahead<T>()
```

где T должен быть типом фиксированного размера. При успешной оценке lookahead возвращает значение типа T. В ParserModel семантика lookahead может выть выражена приведённым ниже псевдокодом.

```
T packet_in.lookahead<T>() {
        bitsToExtract = sizeof(T);
        lastBitNeeded = this.nextBitIndex + bitsToExtract;
        ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
        T tmp = this.data.extractBits(this.nextBitIndex, bitsToExtract);
        return tmp;
}
```

Пример с опциями TCP из параграфа 8.18. Операции над объединениями заголовков также служит иллюстрацией к применению lookahead.

```
state start {
        transition select(b.lookahead<bit<8>>()) {
                0: parse_tcp_option_end;
                1: parse_tcp_option_nop;
```

```
                2: parse_tcp_option_ss;
                3: parse_tcp_option_s;
                5: parse_tcp_option_sack;
        }
}
// некоторые состояния опущены
state parse_tcp_option_sack {
        bit<8> n = b.lookahead<Tcp_option_sack_top>().length;
        b.extract(vec.next.sack, (bit<32>) (8 * n - 16));
        transition start;
}
```

12.8.4. Пропуск битов

P4 обеспечивает два способа пропуска битов во входном пакете без передачи их в заголовок (header). Один способ заключается в извлечение в переменную _, явно задающую тип данных

```
b.extract<T>(_)
```

Другим способом служит использование метода advance, когда число пропускаемых битов известно. В ParserModel метод advance можно выразить приведённым ниже псевдокодом.

```
void packet_in.advance(bit<32> bits) {
        // платформы могут, но не обязаны включать строку проверки
        // verify(bits[2:0] == 0, error.ParserInvalidArgument);
        lastBitNeeded = this.nextBitIndex + bits;
        ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
        this.nextBitIndex += bits;
}
```

12.9. Стеки заголовков

Стеки заголовков имеют два свойства - next и last, которые можно использовать при анализе. Рассмотрим пример, определяющий стек для представления пакетов, содержащих до 10 заголовков MPLS.

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

Выражение mpls.next представляет l-value типа Mpls_h со ссылкой на элемент стека заголовков. Исходно mpls.next указывает первый элемент стека и автоматически перемещается вперёд при каждом успешном извлечении заголовка. Свойство mpls.last указывает элемент, непосредственно предшествующий следующему, если такой элемент существует. Попытка доступа к mpls.next при nextIndex не меньше размера стека вызывает переход в состояние reject и ошибку error.StackOutOfBounds. Аналогично, попытка доступа к mpls.last при nextIndex = 0 вызывает переход в состояние reject и ошибку error.StackOutOfBounds. Ниже приведён пример простого анализатора для обработки MPLS.

```
struct Pkthdr {
        Ethernet_h ethernet;
        Mpls_h[3] mpls;
        // другие заголовки опущены
}

parser P(packet_in b, out Pkthdr p) {
        state start {
                b.extract(p.ethernet);
                transition select(p.ethernet.etherType) {
                        0x8847: parse_mpls;
                        0x0800: parse_ipv4;
                }
        }
        state parse_mpls {
                b.extract(p.mpls.next);
                transition select(p.mpls.last.bos) {
                        0: parse_mpls; // создаёт цикл
                        1: parse_ipv4;
                }
        }
        // другие состояния опущены
}
```

12.10. Субанализаторы

P4 позволяет анализатору использовать другие анализаторы, подобно вызову подпрограмм. Вызываемый субанализатор нужно сначала инициализировать, а затем его экземпляр вызывается с помощью метода apply.

```
parser callee(packet_in packet, out IPv4 ipv4) { /* тело опущено */ }
parser caller(packet_in packet, out Headers h) {
        callee() subparser;
        // экземпляр вызываемого анализатора
        state subroutine {
                subparser.apply(packet, h.ipv4);     // вызов субанализатора
                transition accept;     // если субанализатор завершился состоянием accept
        }
}
```

Семантику вызова субанализатора можно описать следующим образом (рисунок 9):

- состояние, вызывающее субанализатор делится на два полусостояния в операторе вызова анализатора;
- верхняя половина включает переход в состояние start субанализатора;
- состояние субанализатора accept ведёт к переходу в нижнюю половину текущего состояния;
- состояние субанализатора reject переводит текущий анализатор в состояние reject.

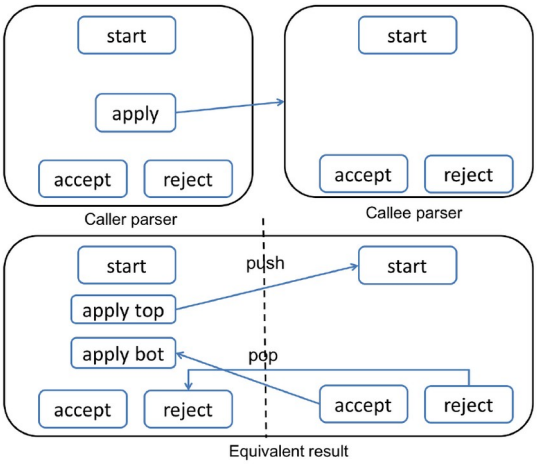

Рисунок 9. Семантика вызова субанализатору - сверху исходная программа, снизу эквивалент.

Поскольку P4 требует объявления перед использованием, невозможно создать (взаимно) рекурсивные анализаторы.

Архитектура может (статически или динамически) ограничивать число состояний, которые анализатор может проходить при разборе пакета. Например, компилятор для конкретной платформы может отвергать анализаторы с циклом, который нельзя развернуть во время компиляции. Если анализатор прерывает работу динамически в результате исчерпания разрешённого для обработки времени, ему следует перейти в состояние reject с ошибкой error.ParserTimeout.

12.11. Набор значений анализатора

В некоторых случаях значения, определяющие переход анализатора из одного состояния в другое, нужно определять во время работы. MPLS является одним из примеров, где поле метки MPLS служит для определения того, что следует за тегом MPLS и это отображение может меняться во время работы. Для поддержки такой функциональности в P4 используется набор значений анализатора (Parser Value Set) - именованный набор значений с API среды выполнения для добавления и удаления элементов. Наборы значений объявляются локально в анализаторе. Значения следует объявлять до ссылки на них в keysetExpression и они могут применяться в выражениях select. Синтаксис объявления показан ниже.

```
valueSetDeclaration
        : optAnnotations
                VALUESET '<' baseType '>' '(' expression ')' name ';'
        | optAnnotations
                VALUESET '<' tupleType '>' '(' expression ')' name ';'
        | optAnnotations
                VALUESET '<' typeName '>' '(' expression ')' name ';'
        ;
```

Наборы значений поддерживают аргумент size, который служит рекомендацией компилятору по резервированию аппаратных ресурсов для набора, например

```
value_set<bit<16>>(4) pvs;
```

создаёт value_set размера 4 с записями типа bit<16>.

Семантика аргумента size похожа на семантику одноимённого свойства таблицы. Если набор значений имеет аргумент size со значением N, это рекомендует компилятору выбрать реализацию плоскости данных, способную сохранять набор из N значений. Набор значений анализатора заполняется плоскостью управления с помощью методов, заданных в спецификации P4Runtime.

13. Блоки управления

Анализаторы P4 отвечают за извлечение битов пакета в заголовки (header), которые вместе с метаданными могут служить для манипуляций блоков управления. Тело блока управления представляет собой традиционную императивную программу. Внутри блока могут вызываться блоки СД для преобразования данных, представляемые в P4 конструкциями, которые называют таблицами.

Синтаксически блок управления задаётся именем, параметрами, необязательными параметрами типа и последовательностью объявления констант, переменных, действий, таблиц и создания других экземпляров.

```
controlDeclaration
        : controlTypeDeclaration optConstructorParameters
                /* controlTypeDeclaration не может включать параметры типа */
                '{' controlLocalDeclarations APPLY controlBody '}'
        ;

controlLocalDeclarations
        : /* пусто */
        | controlLocalDeclarations controlLocalDeclaration
```

```
        ;

controlLocalDeclaration
        : constantDeclaration
        | variableDeclaration
        | actionDeclaration
        | tableDeclaration
        | instantiation
        ;

controlBody
        : blockStatement
        ;
```

В блоках управления не допускается создание экземпляров анализаторов. Описание параметров optConstructorParameters, которые могут применяться для создания параметризованных блоков управления, дано в разделе 14. Параметризация.

В отличие от объявлений типа элемента управления, объявления блоков управления не могут быть базовыми и приведённое ниже объявление является некорректным.

```
control C<H>(inout H data) { /* тело опущено */ }
```

P4 не поддерживает исключений для потока управления внутри элементов управления. Единственным оператором, имеющим нелокальный эффект для потока управления, является оператор exit, незамедлительно прерывающий выполнение содержащего его блока. Т. е. здесь нет эквивалента оператора verify или состояния reject в анализаторах. Поэтому все ошибки должны явно обрабатываться программой.

13.1. Действия

Действиями называют фрагменты кода, которые могут считывать и записывать обрабатываемые данные. Действия могут включать значения данных, которые могут записываться плоскостью управления и считываться плоскостью данных. Действия являются основными конструкциями, с помощью которых плоскость управления может динамически влиять на поведение плоскости данных. Абстрактная модель действия показана на рисунке 10.

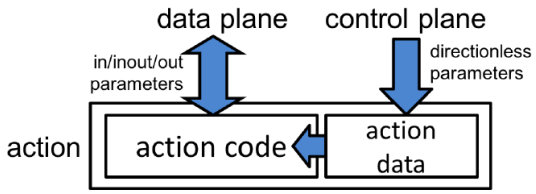

Рисунок 10. Действия включают код (P4) и данные, задаваемые плоскостью управления. Параметры задаются плоскостью данных.

```
actionDeclaration
        : optAnnotations ACTION name '(' parameterList ')' blockStatement
        ;
```

Синтаксически действие похоже на функцию, не возвращающую значения. Действия могут объявляться в блоке управления и в этом случае они доступны лишь в данном экземпляре блока управления. Объявление действия имеет вид

```
action Forward_a(out bit<9> outputPort, bit<9> port) {
        outputPort = port;
}
```

Параметры действия не могут иметь тип extern. Не имеющие направления параметры действия (например, port в предыдущем примере) указывают «данные действия». Все такие параметры должны указываться в конце списка параметров. При использовании в таблице СД (13.2.1.2. Действия) эти параметры будут предоставляться плоскостью управления (например, как указано плоскостью управления, свойством таблицы default_action или свойством const entries в таблице).

Тело действия состоит из операторов и объявлений. Операторы switch не допускаются в действиях - грамматика разрешает их, но семантическим проверкам следует отвергать. Некоторые платформы могут вносить дополнительные ограничения, например, разрешать лишь линейный код без условных операторов и выражений.

13.1.1. Вызов действия

Действия могут выполняться двумя способами.

- Неявно таблицами в процессе обработки СД.
- Явно из блока управления или другого действия. В обоих случаях значения всех параметров действия должны быть заданы явно, включая значения параметров без направления.

13.2. Таблицы

Таблица описывает блоки «сопоставление-действие» (СД). Структура таблицы показана на рисунке 11. Обработка пакета с использованием таблицы СД состоит из нескольких этапов:

- создание ключа;
- поиск ключа в таблице (match), результатом чего является действие (action);
- выполнение действия (action) над входными данными, ведущее к изменению данных.

Объявление table создаёт экземпляр этой таблицы. Для получения нескольких экземпляров таблицы они должны объявляться в блоке управления, создаваемом в нескольких экземплярах.

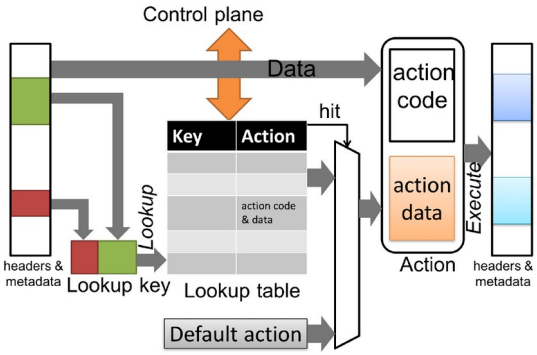

Рисунок 11. Поток данных блока СД.

Таблица поиска является конечным отображением, содержимое которого меняется асинхронно (чтение и запись) плоскостью управления целевой платформы с использованием отдельного API плоскости управления (рисунок 11). Отметим, что термин «таблица» служит как для обозначения таблиц в программах P4, так и внутренних таблиц целевой платформы. Для устранения путаницы иногда будут использоваться термин «блок СД».

Синтаксически таблица определяется в терминах набора пар (свойств) «ключ-значение». Некоторые из этих свойств являются «стандартными», но набор свойств может быть при необходимости расширен компилятором платформы.

```
tableDeclaration
        : optAnnotations TABLE name '{' tablePropertyList '}'
        ;

tablePropertyList
        : tableProperty
        | tablePropertyList tableProperty
        ;

tableProperty
        : KEY '=' '{' keyElementList '}'
        | ACTIONS '=' '{' actionList '}'
        | CONST ENTRIES '=' '{' entriesList '}' /* неизменяемые записи */
        | optAnnotations CONST nonTableKwName '=' initializer ';'
        | optAnnotations nonTableKwName '=' initializer ';'
        ;

nonTableKwName
        : IDENTIFIER
        | TYPE_IDENTIFIER
        | APPLY
        | STATE
        | TYPE
        ;
```

Стандартные свойства таблицы включают:

- ключ (KEY) - выражение, описывающее создание ключа для поиска в таблице;
- действия (ACTIONS) - список всех действий, которые могут быть найдены в таблице.

В дополнение к этому для таблицы могут быть определены следующие свойства:

- действие по умолчанию (default_action), выполняемое в случаях отсутствия в таблице записи для данного ключа;
- размер (size) - целое число, указывающее желаемый размер таблицы.

Компилятор должен устанавливать для по умолчанию действия значение NoAction (а также помещать его в список actions) в таблицах, которые не определяют должным образом свойство default_action (это согласуется с семантикой параграфа 13.2.1.3. Принятое по умолчанию действие). Поэтому все таблицы можно считать имеющими принятое по умолчанию действие, заданное явно или неявно. В этом документе предполагается наличие корректного default_action во всех таблицах.

Таблицы могут также включать зависимые от архитектуры свойства (13.2.1.6. Дополнительные свойства).

Свойства, указанные как const, не могут динамически изменяться плоскостью управления. Свойства key, actions, size всегда являются постоянными, поэтому для них не требуется ключевое слово const.

13.2.1. Свойства таблицы

13.2.1.1. Ключи

Ключ является свойством таблицы, задающим значения плоскости данных, используемые для поиска записей в таблице. Ключ представляет собой список пар (e : m), где e - выражение, описывающее данные для сопоставления с таблицей, а m - константа match_kind, описывающая алгоритм сопоставления (7.1.3. Тип match_kind).

```
keyElementList
        : /* пусто */
        | keyElementList keyElement
        ;

keyElement
```

```
        : expression ':' name optAnnotations ';'
        ;
```

Рассмотрим в качестве примера фрагмент программы

```
table Fwd {
        key = {
                ipv4header.dstAddress : ternary;
                ipv4header.version    : exact;
        }
        // другие поля опущены
}
```

Здесь ключ содержит два поля заголовка ipv4header - dstAddress и version, а match_kind служит трём целям:

- задание алгоритма, используемого для сопоставления данных плоскости данных с записями таблицы в процессе работы;
- создание API плоскости управления для заполнения таблицы;
- использование компилятором back-end при выделении ресурсов для реализации таблицы.

Основная библиотека P4 содержит три предопределённых идентификатора match_kind.

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

Эти идентификаторы соответствуют одноимённым типам сопоставления в $P4_{14}$. Семантика этих типов фактически не требуется для описания поведения абстрактной машины P4, их использование влияет лишь на API плоскости управления и реализацию таблицы поиска. С точки зрения программы P4 таблица поиска является конечным отображением, возвращающим результат в форме действия или индикации отсутствия по заданному ключу, как описано в параграфе 13.2.3. Семантика выполнения блока СД.

Если таблица не имеет свойства key, она не будет таблицей поиска и будет включать лишь принятое по умолчанию действие, т. е. являться пустым отображением.

Каждый элемент ключа может иметь аннотацию @name, используемую для создания видимого плоскости управления имени для поля ключа.

13.2.1.2. Действия

Для таблицы должны объявляться все действия, которые могут присутствовать в связанной с ней таблице поиска или в принятом по умолчанию действии. Это делается с помощью свойства actions, значением которого всегда является actionList.

```
actionList
        : /* пусто */
        | actionList optAnnotations actionRef ';'
        ;

actionRef
        : prefixedNonTypeName
        | prefixedNonTypeName '(' argumentList ')'
;
```

Для иллюстрации вернёмся к примеру VSS из параграфа 5.1. Архитектура VSS.

```
action Drop_action() {
        outCtrl.outputPort = DROP_PORT;
}
action Rewrite_smac(EthernetAddress sourceMac) {
        headers.ethernet.srcAddr = sourceMac;
}
table smac {
        key = { outCtrl.outputPort : exact; }
        actions = {
                Drop_action;
                Rewrite_smac;
        }
}
```

- Записи таблицы smac могут включать два действия - Drop_action и Rewrite_mac.
- Действие Rewrite_smac имеет один параметр - sourceMac, который задаётся плоскостью управления.

Каждое действие в списке actions для таблицы должно иметь своё уникальное имя. Например, приведённый ниже фрагмент будет ошибкой.

```
action a() {}
control c() {
        action a() {}
        // Некорректная таблица - имена двух действий совпадают
        table t { actions = { a; .a; } }
}
```

Каждый параметр с направлением (in, inout, out) должен быть привязан в списке спецификации действий, а параметры без направления не могут быть привязаны к этому списку. Выражения, представленные как аргументы, не оцениваются до вызова действия.

```
action a(in bit<32> x) { /* тело опущено */ }
bit<32> z;
```

```
action b(inout bit<32> x, bit<8> data) { /* тело опущено */ }
table t {
        actions = {
                // a; -- недействительно, параметр x должен быть привязан
                a(5);   // привязка параметра x из a - 5
                b(z);   // привязка параметра x из b - z
                // b(z, 3); недействительная привязка параметра без направления
                // b(); -- недействительно, параметр x должен быть привязан
        }
}
```

13.2.1.3. Принятое по умолчанию действие

Принятое по умолчанию действие вызывается для таблицы блоком СД автоматически, если в таблице не найдено записи для представленного ключа. При наличии свойства default_action оно должно указываться после свойства action. Оно может быть объявлено как константа, что препятствует динамической замене действия плоскостью управления. Действие, принятое по умолчанию, должно быть одним из указанных в списке actions. В частности, выражения, переданные как параметры in, out или inout должны быть синтаксически идентичны выражениям, использованным в одном из элементов списка действий.

Например, для приведённой выше таблицы можно установить неизменное действие по умолчанию

```
const default_action = Rewrite_smac(48w0xAA_BB_CC_DD_EE_FF);
```

Отметим, что заданное действие по умолчанию должно предоставлять аргументы для привязанных плоскостью управления параметров (параметры без направления), поскольку принятое по умолчанию действие синтезируется во время компиляции. Выражения, представленные как аргументы для параметров с направлениям (in, inout, out), оцениваются при вызове действия, а выражения для параметров без аргументов - во время компиляции.

В продолжение примера из предыдущего параграфа здесь приведено несколько корректных и некорректных спецификаций принятых по умолчанию действий для таблицы t.

```
default_action = a(5); // корректно, нет параметров плоскости управления
// default_action = a(z); -- некорректно, параметр x для a уже привязан к значению 5
default_action = b(z,8w8); // корректно, параметр data для b привязан к 8w8
// default_action = b(z); -- некорректно, параметр data для b не привязан
// default_action = b(x, 3); -- некорректно, параметр x для b привязан к x вместо z
```

Если таблица не задаёт свойство default_action и пакету не соответствует никакая запись, таблица не влияет на пакет и обработка продолжается в соответствии с императивным потоком управления программы.

13.2.1.4. Записи

Хотя записи таблиц обычно создаются плоскостью управления, возможна инициализация таблиц во время компиляции с использованием набора записей. Это полезно в ситуациях, где таблицы служат для реализации фиксированных алгоритмов - статическое задание таблиц позволяет задать эти алгоритмы в P4, а это даёт компилятору возможность понять реальное использование таблицы и принять более эффективное решение о выделении ограниченных ресурсов платформы. Записи, объявленные в коде P4, включаются в таблицу при загрузке программы на целевой платформе. Синтаксис определения записей таблиц показан ниже.

```
tableProperty
        : const ENTRIES '=' '{' entriesLlist '}' /* неизменные записи */

entriesList
        : entry
        | entriesList entry
        ;

entry
        : keysetExpression ':' actionRef optAnnotations ';'
        ;
```

Записи таблиц неизменны (const), т. е. могут лишь считываться и плоскость управления не может изменить или удалить их. Из этого следует неизменность таблиц, определяющих записи в коде P4. Такой выбор имеет важное влияние на среду выполнения P4, поскольку не нужно отслеживать различные типы записей в таблице (изменяемые и статические). В будущих версиях P4 может быть добавлена возможность смешивать изменяемые и статические записи в одной таблице путём объявления дополнительных свойств записей без ключевого слова const.

Компонент записи keysetExpression является кортежем, который должен обеспечивать поле для каждого ключа в таблице (13.2.1. Свойства таблицы). Тип ключа таблицы должен соответствовать типу элемента в наборе. Компонент actionRef должен быть действием, которое присутствует в списке действий таблицы со всеми привязанными аргументами.

Если API среды выполнения требует приоритета для записей таблицы (например, при использовании P4 Runtime API таблицы хотя бы с одним троичным полем ключа поиска), записи сопоставляются в порядке указания в программе с остановкой на первой совпадающей записи. Архитектуре следует определять значимость порядка записей (если он имеется) для других типов таблиц.

В зависимости от match_kind для ключей выражения набора ключей могут задавать одну или множество записей. Компилятор будет синтезировать нужное число записей для установки в таблицу. Свойства платформы могут дополнительно ограничивать возможности синтезирования записей. Например, если число синтезируемых записей превосходит размер таблицы, реализация компилятора может выдавать предупреждение или ошибку в зависимости от возможностей целевой платформы.

Для иллюстрации рассмотрим приведённый ниже пример.

```
header hdr {
        bit<8> e;
        bit<16> t;
```

```
        bit<8> l;
        bit<8> r;
        bit<1> v;
}

struct Header_t {
        hdr h;
}

struct Meta_t {}

control ingress(inout Header_t h, inout Meta_t m,
                inout standard_metadata_t standard_meta) {
        action a() { standard_meta.egress_spec = 0; }
        action a_with_control_params(bit<9> x) { standard_meta.egress_spec = x; }

        table t_exact_ternary {
                key = {
                        h.h.e : exact;
                        h.h.t : ternary;
                }
                actions = {
                        a;
                        a_with_control_params;
                }
                default_action = a;
                const entries = {
                        (0x01, 0x1111 &&& 0xF) : a_with_control_params(1);
                        (0x02, 0x1181) : a_with_control_params(2);
                        (0x03, 0x1111 &&& 0xF000) : a_with_control_params(3);
                        (0x04, 0x1211 &&& 0x02F0) : a_with_control_params(4);
                        (0x04, 0x1311 &&& 0x02F0) : a_with_control_params(5);
                        (0x06, _) : a_with_control_params(6);
                        _         : a;
                }
        }
}
```

Здесь определён набор из 7 записей, каждая из которых вызывает действие a_with_control_params, за исключением последней, которая вызывает действие a. После загрузки программы эти записи устанавливаются в таблице в порядке их указания в программе.

13.2.1.5. Размер

Свойство size является дополнительным для таблицы и при наличии этого свойства его значение всегда является целым числом, известным при компиляции. Значение указывает число записей в таблице.

Если у таблицы имеется значение размера N, компилятору рекомендуется выбрать реализацию плоскости данных, способную сохранять в таблице N записей. Это не гарантирует вставку в таблицу N произвольных записей и означает лишь возможность размещения в таблице некого набора из N записей. Например, попытка поместить в таблицу некоторую комбинацию из N записей может привести к отказу по причине того, что компилятор выбрал хэш-таблицу с гарантированным временем поиска O(1).

Если реализация P4 должна оценивать ресурсы таблицы во время компиляции, таблица со свойством size может быть сочтена ошибкой. Некоторые реализации P4 могут быть способны динамически менять размер таблицы в процессе работы. Если значение size задано в программе P4, таким реализациям рекомендуется использовать это значение как начальный размер таблицы.

13.2.1.6. Дополнительные свойства

Объявление таблицы определяет важные интерфейсы плоскостей управления и данных - ключи и действия. Однако лучший способ реализации таблицы на деле может зависеть от природы записей, создаваемых в процессе работы (например, таблица может быть плотной или редкой, может быть реализована как хэш-таблица, ассоциативная память, дерево и т. п.). Кроме того, архитектура может поддерживать дополнительные свойства, семантика которых выходит за рамки данной спецификации. Например, в архитектуре со статическим выделением ресурсов таблиц может потребоваться задание в программе свойства size, которое компилятор back-end может использовать для выделения ресурсов хранения. Однако такие зависимые от архитектуры свойства не могут менять семантику поиска в таблицах, который всегда находит нужное действие или отсутствие такового. Можно лишь изменить интерпретацию результата поиска в плоскости данных. Это ограничение нужно для того, чтобы обеспечить возможность понять поведение таблиц во время компиляции.

В качестве другого примера свойство реализации можно использовать для передачи дополнительной информации компилятору back-end. Значение этого свойства может быть экземпляром внешнего блока, выбранного из подходящей библиотеки. Например, базовая функциональность конструкции action_profile в таблице P4[14] может быть реализована, как показано ниже.

```
extern ActionProfile {
        ActionProfile(bit<32> size); // число предполагаемых разных действий
}
table t {
        key = { /* тело опущено */ }
        size = 1024;
        implementation = ActionProfile(32);  // вызов конструктора
}
```

Здесь можно использовать профиль действия для оптимизации, если таблица имеет много записей, но предполагается, что действий, связанных с этими записями будет немного. Добавление уровня опосредованности позволяет совместно использовать идентичные записи, что может существенно снижать требования к хранилищу.

13.2.2. Вызов блока СД

Таблицу можно вызвать с помощью метода apply. Вызов этого метода для экземпляра таблицы возвращает значение типа struct с двумя полями. Эта структура создаётся компилятором автоматически. Для каждой таблицы T компилятор синтезирует enum и struct, как показано ниже.

```
enum action_list(T) {
        // одно поле для каждого действия из списка action в таблице T
}
struct apply_result(T) {
        bool hit;
        action_list(T) action_run;
}
```

Оценка метода apply устанавливает в поле hit значение true, а в miss - false, если в таблице найдено совпадение. В противном случае устанавливается hit = false и miss = true. Эти биты могут применяться в потоке управления вызвавшего таблицу блока управления.

```
if (ipv4_match.apply().hit) {
        // найдено совпадение (hit)
} else {
        // совпадения не найдено (miss)
}
if (ipv4_host.apply().miss) {
        ipv4_lpm.apply(); // поиск маршрута при отсутствии записи в таблице host
}
```

Поле action_run показывает тип выполняемого действия (независимо от hit или miss) и может использоваться в операторе switch, как показано ниже.

```
switch (dmac.apply().action_run) {
        Drop_action: { return; }
}
```

13.2.3. Семантика выполнения блока СД

Семантика оператора вызова таблицы показана ниже

```
m.apply();
```

и может быть представлена показанным ниже псевдокодом (см. рисунок 11).

```
apply_result(m) m.apply() {
        apply_result(m) result;
        var lookupKey = m.buildKey(m.key); // использование блока ключей
        action RA = m.table.lookup(lookupKey);
        if (RA == null) {      // нет в таблице поиска (miss)
                result.hit = false;
                RA = m.default_action;// используется принятое по умолчанию действие
        } else {
                result.hit = true;
        }
        result.action_run = action_type(RA);
        evaluate_and_copy_in_RA_args(RA);
        execute(RA);
        copy_out_RA_args(RA);
        return result;
}
```

Вызов buildKey в приведённом выше псевдокоде оценивает каждое выражение key, чтобы понять, присутствует ли ключ в определении ключей таблицы. Поведение должно быть таким, как будто результат оценки каждого выражения назначен свежей временной переменной, перед началом оценки следующего выражения. Ниже приведён пример определения таблицы P4 и вызова apply.

```
bit<8> f1 (in bit<8> a, inout bit<8> b) {
        b = a + 5;
        return a >> 1;
}
bit<8> x;
bit<8> y;
table t1 {
        key = {
                y & 0x7 : exact @name("masked_y");
                f1(x, y)        : exact @name("f1");
                y               : exact;
        }
        // ... здесь определяются остальные свойства таблицы, на связанные с примером
}
apply {
        // здесь присваиваются значения x и y, на связанные с примером
        t1.apply();
}
```

Это эквивалентно поведению другого определения таблицы и вызову apply.

```
// такие же определения f1, x, y как в предыдущем примере
bit<8> tmp_1;
```

```
bit<8> tmp_2;
bit<8> tmp_3;
table t1 {
        key = {
                tmp_1 : exact @name("masked_y");
                tmp_2 : exact @name("f1");
                tmp_3 : exact @name("y");
        }
        // ... здесь определяются остальные свойства таблицы, на связанные с примером
}
apply {
        // здесь присваиваются значения x и y, на связанные с примером
        tmp_1 = y & 0x7;
        tmp_2 = f1(x, y);
        tmp_3 = y;
        t1.apply();
}
```

Отметим, что второй пример приведён для задания поведения первого примера. Реализация может выбрать любой вариант, обеспечивающий такое поведение[1].

13.3. Абстрактная машина конвейера СД

Можно описать вычислительную модель конвейера СД, воплощённого в блоке управления - тело блока выполняется аналогично традиционным императивным программам:

- в среде выполнения операторы блока применяются в порядке их следования;
- выполнение оператора return незамедлительно прерывает исполнение текущего блока и возвращает управления в точку вызова;
- выполнение оператора exit незамедлительно прерывает исполнение текущего блока и всех блоков, в которых он содержится (откуда вызван);
- использование таблицы выполняет соответствующий блок СД, как описано выше.

13.4. Вызов элемента управления

P4 позволяет элементам управления обращаться к услугам других элементов управления, вызывая их подобно подпрограммам. Для вызова другого элемента управления нужно сначала создать его экземпляр, который потом вызывается с помощью метода apply, как показано ниже.

```
control Callee( inout IPv4 ipv4) { /* тело опущено */ }
control Caller(inout Headers h) {
        Callee() instance;     // экземпляр вызываемого блока
        apply {
                instance.apply(h.ipv4);       // вызов блока управления
        }
}
```

14. Параметризация

Для поддержки библиотек полезных компонентов P4 анализаторы и блоки управления можно параметризовать с помощью параметров конструктора. Рассмотрим синтаксис объявления конструктора

```
parserDeclaration
        : parserTypeDeclaration optConstructorParameters
                '{' parserLocalElements parserStates '}'
        ;

optConstructorParameters
        : /* пусто */
        | '(' parameterList ')'
        ;
```

Из этого правила можно вывести возможность наличия у конструктора двух наборов параметров:

- параметры среды выполнения (parameterList);
- необязательные параметры конструктора (optConstructorParameters).

Параметры конструктора должны быть ненаправленными (не могут быть in, out, inout) и при создании экземпляра анализатора должна быть возможность полной оценки выражений, представленных для этих параметров, во время компиляции. Рассмотрим пример

```
parser GenericParser(packet_in b, out Packet_header p)
                    (bool udpSupport) {    // параметры конструктора
        state start {
                b.extract(p.ethernet);
                transition select(p.ethernet.etherType) {
                        16w0x0800: ipv4;
                }
        }
        state ipv4 {
                b.extract(p.ipv4);
```

[1] *В большинстве программ $P4_{16}$ не применяются вызовы функций или методов в выражениях для ключей таблицы и порядок оценки выражений не влияет на результирующее значение ключа поиска. В распространённом случае, когда реализация помещает дополнительные операторы присваивания для реализации выражений с ключами, имеющих побочные эффекты, но не использует такие операторы для выражений без побочных эффектов, такие операторы практически не применяются.*

```
                transition select(p.ipv4.protocol) {
                        6: tcp;
                        17: tryudp;
                }
        }
        state tryudp {
                transition select(udpSupport) {
                        false: accept;
                        true : udp;
                }
        }
        state udp {
                // тело опущено
        }
}
```

При создании GenericParser нужно представить значение для параметра udpSupport, как в примере ниже.

```
// topParser - это GenericParser, где udpSupport = false
GenericParser(false) topParser;
```

14.1. Прямой вызов типа

Экземпляры элементов управления и анализаторов зачастую создаются однократно. Объявления элементов управления и анализаторов без конструктора могут применяться напрямую, как будто экземпляр уже есть. Это создаёт и применяет локальный экземпляр данного типа.

```
control Callee(/* параметры опущены */) { /* тело опущено */ }
control Caller(/* параметры опущены */)(/* параметры опущены */) {
        apply {
                Callee.apply(/* аргументы опущены */); // callee считается экземпляром
        }
}
```

Определение Caller эквивалентно приведённому ниже.

```
control Caller(/* параметры опущены */)(/* параметры опущены */) {
        @name("Callee") Callee() Callee_inst; // локальный экземпляр Callee
        apply {
                Callee_inst.apply(/* аргументы опущены */); // применение Callee_inst
        }
}
```

Это свойство предназначено для упрощения общего случая, где экземпляр типа создаётся однократно. Для полноты поведение неоднократного вызова того же типа напрямую определено в соответствии с приведённым описанием.

- Прямой вызов типа в разных областях действия ведёт к созданию разных локальных экземпляров с различающимися полными именами элементов управления.
- В одной области действия прямые вызовы типа приводят к созданию своего локального экземпляра для каждого вызова, однако однотипные экземпляры будут иметь одно глобальное имя через аннотацию @name (17.3.2. Аннотации, управляющие именами). Если тип содержит управляемые элементы, неоднократные непосредственные вызовы в одной области действия недопустимы, поскольку будут создаваться несколько экземпляров управляемых элементов с одним именем.

15. Сборка пакета

Сборка пакетов (deparsing) в каком-то смысле обратна их анализу. P4 не поддерживает отдельного языка для сборки пакетов и она выполняется в блоках управления, имеющих по меньшей мере один параметр packet_out. Приведённый ниже фрагмент кода записывает поочерёдно заголовки Ethernet и IPv4 в packet_out.

```
control TopDeparser(inout Parsed_packet p, packet_out b) {
        apply {
                b.emit(p.ethernet);
                b.emit(p.ip);
        }
}
```

Выдача заголовка добавляет header к packet_out, если этот заголовок действителен. Выдача стека заголовков будет добавлять все элементы стека заголовков в порядке роста индексов.

15.1. Вставка данных в пакет

Тип packet_out определён в основной библиотеке P4 и это определение приведено ниже. Тип обеспечивает метод добавления данных в выходной пакет, называемый emit:

```
extern packet_out {
        void emit<T>(in T data);
}
```

Метод emit поддерживает добавления данных в заголовок, стек или объединение заголовков для выходного пакета.

- При использовании с заголовком emit добавляет данные в заголовок пакета, если этот заголовок действителен, и ничего не делает в противном случае (no-op).
- При использовании со стеком заголовков emit вызывается рекурсивно для каждого элемента стека.
- При использовании со структурой или объединением заголовков emit рекурсивно вызывается для каждого поля. Отметим, что в struct не допускаются поля типа error и enum, поскольку они не сериализуются.

Недопустим вызов emit для выражений базового типа, enum или error.

Вызов emit можно описать приведённым ниже псевдокодом.

```
packet_out {
        byte[] data;
        unsigned lengthInBits;
        void initializeForWriting() {
                this.data.clear();
                this.lengthInBits = 0;
        }
        /// Добавляются данные в пакет. T может быть заголовком, стеком или
        /// объединением заголовков и структурой из таких типов.
        void emit<T>(T data) {
                if (isHeader(T))
                        if (data.valid$) {
                                this.data.append(data);
                                this.lengthInBits += data.lengthInBits;
                        }
                else if (isHeaderStack(T))
                        for (e : data)
                                emit(e);
                else if (isHeaderUnion(T) || isStruct(T))
                        for (f : data.fields$)
                                emit(e.f)
                // Другие типы T недействительны
        }
}
```

Здесь применяются специальные идентификаторы valid$ для скрытого бита validity в заголовках и fields$ для списка полей struct или объединения заголовков. Применяется также стандартная нотация для итераций по элементам стека (e : data) и списку полей объединений заголовков или struct (f : data.fields$). Для struct итерации выполняются в порядке указания полей при определении типа.

16. Описание архитектуры

Описание архитектуры должно предоставляться производителем платформы в форме исходного кода библиотеки P4, который содержит по меньшей мере одно объявление пакета. Экземпляр этого пакета пользователь должен создать для своей программы. Примером может служить описание VSS в параграфе 5.1. Архитектура VSS.

Файл описания архитектуры может определять типы данных, константы, реализации вспомогательных программ (package) и ошибки. Он должен объявлять типы всех программируемых блоков, которые могут появляться на целевой платформе (анализаторы и блоки управления). Программируемые блоки могут группироваться в пакеты (package), которые могут быть вложенными.

Поскольку некоторые компоненты платформ могут манипулировать заданными пользователем данными, которые неизвестны в момент создания файла описания, они описываются с помощью переменных типа, которые должны параметрически использоваться в программе (т. е. тип переменной проверяется, подобно универсальным типам Java, а не шаблонам C++).

16.1. Пример описания архитектуры

Приведённый ниже пример описывает коммутатор, использующий два пакета, каждый из которых содержит анализатор, конвейер СД и сборщик.

```
parser Parser<IH>(packet_in b, out IH parsedHeaders);
// входной конвейер СД
control IPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         out OH outputHeaders,
                         out T toEgress,
                         out OutControl outCtrl);
// выходной конвейер СД
control EPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         in T fromIngress,
                         out OH outputHeaders,
                         out OutControl outCtrl);
control Deparser<OH>(in OH outputHeaders, packet_out b);
package Ingress<T, IH, OH>(Parser<IH> p,
                           IPipe<T, IH, OH> map,
                           Deparser<OH> d);
package Egress<T, IH, OH>(Parser<IH> p,
                          EPipe<T, IH, OH> map,
                          Deparser<OH> d);
package Switch<T>(Ingress<T, _, _> ingress, Egress<T, _, _> egress);
```

Из этих объявлений можно получить полезную информацию об архитектуре описываемого коммутатора даже без чтения точного описания архитектуры (рисунок 12).

- Коммутатор содержит два отдельных пакета Ingress и Egress.

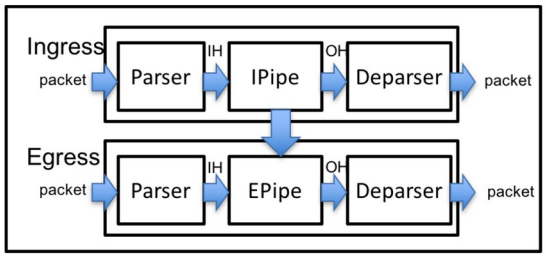

Рисунок 12. Фрагмент примера коммутатора.

- Блоки Parser, IPipe, Deparser в пакете Ingress соединены в цепочку. Кроме того, блок Ingress.IPipe имеет ввод типа Ingress.IH, который является выводом Ingress.Parser.
- Аналогично пакет Egress включает блоки Parser, EPipe, Deparser.
- Ingress.IPipe соединён с Egress.EPipe, поскольку первый даёт на выход значение типа T, которое служит вводом для второго. Отметим, что экземпляры типа T создаются также в пакете Switch. Напротив, входной (Ingress) тип IH и выходной (Egress) тип IH могут различаться. Чтобы они совпадали, можно вместо объявления IH и OH на уровне коммутатора задать

```
package Switch<T,IH,OH>(Ingress<T, IH, OH> ingress, Egress<T, IH, OH> egress).
```

Эта архитектура моделирует коммутатор, содержащий два разных канала между входным и выходным конвейером.

- Канал передачи данных непосредственно через аргумент типа T. На программной платформе с общей памятью для входного и выходного конвейера это можно реализовать путём передачи указателя, но на платформах без общей памяти компилятор должен автоматически синтезировать код сериализации.
- Канал опосредованной передачи с использованием анализатора с сериализацией данных в пакет и обратно.

16.2. Пример программы для архитектуры

Для работы на определённой архитектуре программа P4 должна создать экземпляр пакета верхнего, уровня передавая значения для всех его аргументов и создавая переменную с именем main в пространстве имён верхнего уровня. Типы аргументов должны соответствовать типам параметров после подходящей подстановки типов переменных. Подстановка типа может быть выражена напрямую с использованием специализации типа или выведена компилятором с использованием алгоритма унификации, подобного Hindley-Milner. Например, с объявлениями типов

```
parser Prs<T>(packet_in b, out T result);
control Pipe<T>(in T data);
package Switch<T>(Prs<T> p, Pipe<T> map);
```

и следующими объявлениями

```
parser P(packet_in b, out bit<32> index) { /* тело опущено */ }
control Pipe1(in bit<32> data) { /* тело опущено */ }
control Pipe2(in bit<8> data) { /* тело опущено */ }
```

Ниже приведено действительное объявление для целевой платформы верхнего уровня

```
Switch(P(), Pipe1()) main;
```

Следующее определение недействительно

```
Switch(P(), Pipe2()) main;
```

поскольку анализатор P требует для T тип bit<32>, а Pipe2 требует от T тип bit<8>.

Пользователь может явно задать значения переменных типа (иначе компилятор выведет их

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

16.3. Модель фильтра пакетов

Для демонстрации универсальности языка описания архитектуры P4 рассмотрим пример фильтрации пакетов, где решения принимаются исключительно по результатам анализатора P4, как показано на рисунке 13.

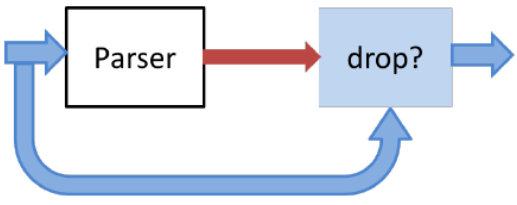

Рисунок 13. Модель фильтра пакетов.

Эту модель можно применить для фильтрации пакетов в ядре Linux. Например, можно заменить язык tcpdump более мощным языком P4, что позволит поддерживать новые протоколы с обеспечением полной «безопасности типов» при обработке пакетов. Для такой платформы компилятор P4 может генерировать программу eBPF (Extended Berkeley Packet Filter), которая инжектируется утилитой tcpdump в ядро Linux и выполняется EBPF kernel JIT.

Для цели в виде ядра Linux и архитектурной модели фильтра пакетов можно объявить

```
parser Parser<H>(packet_in packet, out H headers);
control Filter<H>(inout H headers, out bool accept);

package Program<H>(Parser<H> p, Filter<H> f);
```

17. Абстрактная машина P4 - оценка

Оценка программы P4 выполняется в два этапа:

- статическая оценка происходит во время компиляции программы P4 путём анализа и создания экземпляров всех блоков с состоянием;
- динамическая оценка происходит в среде выполнения путём изолированного выполнения каждого функционального блока P4, когда он получает управление от архитектуры.

17.1. Известные при компиляции значения

- Целочисленные, логические и строковые литералы.
- Идентификаторы из объявления error, enum, match_kind.
- Идентификатор default.
- Поле size значения с типом стека заголовков.
- Идентификатор _ при использовании в выражениях select.
- Идентификаторы, представляющие объявленные типы, действия, таблицы, анализаторы, элементы управления, пакеты.
- Списки, в которых все компоненты имеют известные при компиляции значения.
- Выражения инициализации структуру, где все поля известны при компиляции.
- Экземпляры, создаваемые объявлениями (10.3. Создание экземпляров) и вызовами конструкторов.
- Выражения +, -, *, / , %, cast, !, &, |, &&, ||, << , >> , ~ , >, <, ==, !=, <=, >=, ++, [:], где все операнды известны при компиляции.
- Идентификаторы, объявленные как константы с использованием ключевого слова const.
- Выражения в форме e.minSizeInBits() и e.minSizeInBytes().

17.2. Оценка при компиляции

Оценка программы выполняется в порядке объявлений, начиная с пространства имён верхнего уровня.

- Все объявления (например, анализаторы, элементы управления, типы, константы) оценивают сами себя.
- Для каждой таблицы оценивается экземпляр.
- Вызовы конструкторов оцениваю объекты с состоянием соответствующего типа. Для этого все аргументы конструктора оцениваются рекурсивно и привязываются к параметрам конструктора. Аргументы конструктора должны быть известны при компиляции. Порядок оценки аргументов конструктора не должен играть роли, поскольку он не влияет на результат.
- Экземпляры оценивают именованные объекты с состоянием.
- Экземпляр анализатора или элемента управления рекурсивно оценивает все экземпляры с состоянием, объявленные в блоке.
- Результатом оценки программы является значение переменной верхнего уровня main.

Отметим, что все значения с состоянием оцениваются во время компиляции.

В качестве примера рассмотрим фрагмент программы, приведённый ниже.

```
// объявления архитектуры
parser P(/* параметры опущены */);
control C(/* параметры опущены */);
control D(/* параметры опущены */);

package Switch(P prs, C ctrl, D dep);
extern Checksum16 { /* тело опущено */}

// пользовательский код
Checksum16() ck16; // экземпляр блока контрольных сумм
parser TopParser(/* параметры опущены */)(Checksum16 unit) { /* тело опущено */}
control Pipe(/* параметры опущены */) { /* тело опущено */}
control TopDeparser(/* параметры опущены */)(Checksum16 unit) { /* тело опущено */}

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

Оценка этой программы происходит в описанном ниже порядке.

1. Объявления P, C, D, Switch, Checksum16 оценивают себя сами.
2. Экземпляр Checksum16() ck16 оценивается и создаёт объект ck16 типа Checksum16.
3. Объявления TopParser, Pipe, TopDeparser оценивают себя сами.
4. Выполняется оценка экземпляра переменной :
   (a) рекурсивно оцениваются аргументы конструктора;
   (b) вызывается конструктор TopParser(ck16)
   (c) аргументы оцениваются рекурсивно, оценивается объект ck16;

(d) оценивается сам конструктор, что ведёт к созданию объекта типа TopParser;

(e) аналогично Pipe() и TopDeparser(ck16) оцениваются как вызовы конструктора;

(f) оцениваются все аргументы конструктора пакета Switch (экземпляры TopParser, Pipe, TopDeparser) и их сигнатуры сопоставляются с определением Switch;

(g) оценивается конструктор Switch, результатом чего является экземпляр пакет Switch (prs типа TopParser является первым параметром, ctrl типа Pipe - вторым, dep типа TopDeparser - третьим).

5. Результатом оценки программы является значение переменной main - экземпляра пакета Switch.

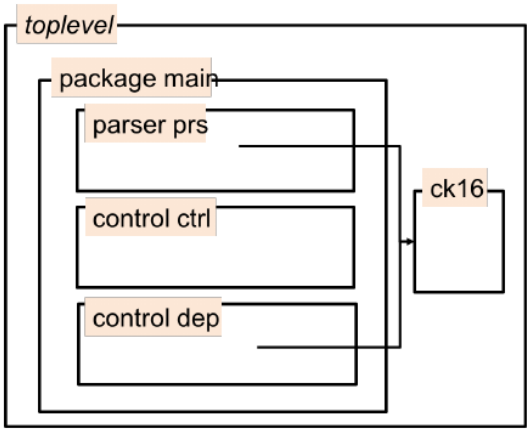

Рисунок 14. Результат оценки.

На рисунке 14 показан результат оценки в графической форме, представляющий собой граф экземпляров. Имеется 1 экземпляр Checksum16 (ck16), совместно используемый TopParser и TopDeparser (конкретная архитектура может потребовать использования разных блоков контрольной суммы).

17.3. Имена элементов управления

Каждый управляемый объект, раскрываемый программой P4, должен иметь уникальное полное имя, которое плоскость управления может использовать для взаимодействия с объектом. Управляемые объекты включают таблицы, ключи, действия, экземпляры extern. Полное имя состоит из локального имени управляемого объекта с префиксом из имени включающего его пространства имён. Поэтому программные конструкции с элементами управления также должны иметь уникальные полные имена. Это экземпляры элементов управления и анализаторов. Оценка может создавать множество однотипных экземпляров, каждый из которых должен иметь уникальное полное имя.

17.3.1. Вычисление имён элементов управления

Полное имя конструкции создаётся конкатенацией полного имени включающего её блока с локальным именем. Конструкции без включающего пространства имён (т. е. определённые глобально) имеют одинаковые локальное и глобальное имена. Локальны имена управляемых объектов и включающих их конструкций выводятся из синтаксиса программы P4, как описано ниже.

17.3.1.1. Таблицы

Для каждой конструкции table её синтаксическое имя становится локальным именем таблицы. Например, определение

```
control c(/* параметры опущены */)() {
        table t { /* тело опущено */ }
}
```

создаст таблицу с локальным именем t.

17.3.1.2. Ключи

Синтаксически ключи таблицы являются выражениями. Для простых выражений локальное имя может создаваться из самого выражения. Ниже приведён пример таблицы t, ключи которой названы data.f1 и hdrs[3].f2.

```
table t {
        keys = {
                data.f1 : exact;
                hdrs[3].f2 : exact;
        }
        actions = { /* тело опущено */ }
}
```

Перечисленные в таблице виды выражений имеют локальные имена, выведенные из их синтаксических имён.

Выражение	*Пример*	*Имя*
Метод isValid()	h.isValid()	"h.isValid()"
Доступ к массиву	header_stack[1]	"header_stack[1]"
Константа	1	"1"
Проекция поля	data.f1	"data.f1"
Нарезка	F1[3:0]	"f1[3:0]"

Все прочие выражения должны аннотироваться с использованием @name (18.3.3. Аннотации API плоскости управления), как показано ниже.

```
table t {
        keys = {
                data.f1 + 1 : exact @name("f1_mask");
```

```
        }
        actions = { /* тело опущено */ }
}
```

Здесь аннотация @name("f1_mask") назначает ключу локальное имя "f1_mask".

17.3.1.3. Действия

Для конструкции action локальным именем действия является синтаксическое имя action. Например,

```
control c(/* параметры опущены */)() {
        action a(...) { /* тело опущено */ }
}
```

создаёт локальное имя a.

17.3.1.4. Экземпляры

Локальные имена экземпляров extern, parser и control выводятся на основе использования экземпляра. Если экземпляр привязан к имени, оно становится локальным именам для плоскости управления. Например, при объявлении control C

```
control C(/* параметры опущены */)() { /* тело опущено */ }
```

и создании экземпляра

```
C() c_inst;
```

локальным именем будет c_inst. Если экземпляр создаётся в качестве аргумента, его локальным именем будет имя формального параметра, к которому экземпляр привязан. Например при объявлении extern E и control C

```
extern E { /* тело опущено */ }
control C( /* параметры опущены */ )(E e_in) { /* тело опущено */ }
```

и создании экземпляра

```
C(E()) c_inst;
```

локальным именем экземпляра extern будет e_in.

Если создаваемая конструкция передаётся как аргумент пакету (package), имя экземпляра выводится из представленного пользователем объявления, когда это возможно. В приведённом ниже примере локальным именем MyC будет c, а локальным именем extern - e2, а не e1.

```
extern E { /* тело опущено */ }
control ArchC(E e1);
package Arch(ArchC c);

control MyC(E e2)() { /* тело опущено */ }
Arch(MyC()) main;
```

Отметим, что в этом примере архитектура будет представлять экземпляр extern при передаче экземпляра MyC пакету Arch. Полным именем этого экземпляра будет main.c.e2.

Далее рассмотрим более крупный пример, показывающий генерацию имён при наличии множества экземпляров.

```
control Callee() {
        table t { /* тело опущено */ }
        apply { t.apply(); }
}
control Caller() {
        Callee() c1;
        Callee() c2;
        apply {
                c1.apply();
                c2.apply();
        }
}
control Simple();
package Top(Simple s);
Top(Caller()) main;
```

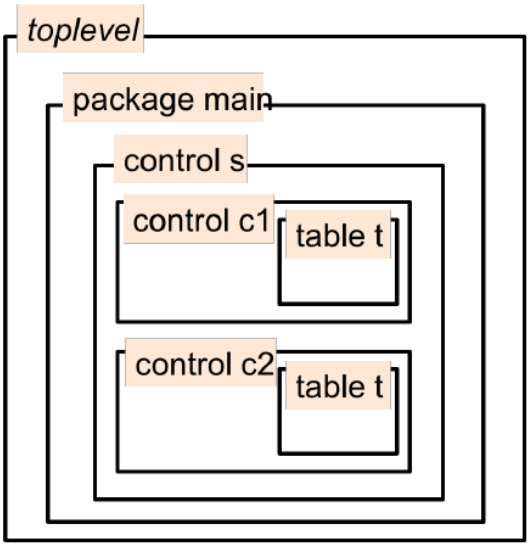

Рисунок 15. Оценка программы с несколькими экземплярами компонентов.

Оценка этой программы при компиляции создаёт структуру, показанную на рисунке 15. Отметим наличие двух экземпляров таблицы t, которые (оба) должны быть раскрыты плоскости управления. Для именования объектов в этой иерархии используется компонент пути имён, содержащих экземпляры. В данном случае две таблицы будут называться s.c1.t и s.c2.t, где s - имя аргумента для создания экземпляра пакета, которое выводится из имени соответствующего формального параметра.

17.3.2. Аннотации, управляющие именами

Аннотации, относящиеся к плоскости управления (18.3.3. Аннотации API плоскости управления), могут менять видимые плоскости управления имена.

- Аннотация @hidden скрывает управляемый элемент от плоскости управления. Это единственный случай, когда управляемому объекту не требуется уникальное полное имя.
- Можно использовать аннотацию @ для смены локального имени управляемого объекта.

Программы, выдающие одно полное имя для двух разных управляемых элементов, недопустимы.

17.3.3. Рекомендации

Плоскость управления может указывать управляемый объект суффиксом его полного имени, если это обеспечивает однозначность в контексте применения. Рассмотрим пример.

```
control c( /* параметры опущены */)() {
        action a ( /* параметры опущены */ ) { /* тело опущено */ }
        table t {
                keys = { /* тело опущено */ }
                actions = { a; }
        }
}
c() c_inst;
```

Программы плоскости управления могут указывать действие c_inst.a как a при вставке правил в таблицу c_inst.t, поскольку это ясно из определения таблица, к которой относится действие.

Не все однозначные сокращения можно рекомендовать. Рассмотрим первый пример из параграфа 17.3. Имена элементов управления. Можно подумать о ссылке на s.c1 как c1, поскольку в программе нет другого объекта c1. Однако это сделает программу «хрупкой», поскольку её новые версии не смогут создавать экземпляр с именем c1 или включать библиотеки P4, где имеется объект с таким именем.

17.4. Динамическая оценка

Динамическая оценка программу P4 организуется архитектурной моделью. Каждая модель должна задавать порядок и условия динамического выполнения различных компонентов программы P4. Например, в VSS из раздела 5.1. Архитектура VSS поток выполнения имеет вид Parser->Pipe->Deparser. При вызове блока исполнения P4 он работает до завершения (прерывания) в соответствии с описанной здесь семантикой.

17.4.1. Модель одновременной работы

Типичной системе обработки пакетов требуется одновременно выполнять множество логических потоков (thread). По меньшей мере имеется поток, выполняемый плоскостью управления и меняющий содержимое таблиц. Спецификации архитектуры следует подробно описывать взаимодействия между плоскостями управления и данных. Плоскость данных может обмениваться с плоскостью управления через вызовы внешних методов и функций. Высокоскоростные системы обрабатывают множество пакетов одновременно (например, в конвейерах) или анализируют пакет одновременно с выполнением операций СД для другого пакета. В этом параграфе описана семантика программ P4 в части одновременной работы.

Каждый анализатор или блок управления верхнего уровня выполняется в форме отдельного потока (thread) создаваемого архитектурой. Все параметры и локальные переменные блока доступны лишь этому потоку. Это относится к параметрам анализаторов и сборщиков packet_in и packet_out. Поскольку блок P4 использует лишь свои

локальные ресурсы (метаданные, заголовки, переменные), его поведение при одновременной работе не отличается от изолированного поведения и чередование операторов из разных блоков не должно влиять на результаты.

Внешние блоки, экземпляры которых создаются программой P4, являются глобальными и используются всеми потоками. Если внешние блоки участвуют в доступе к состоянию (счётчики, регистры), т. е. к методам чтения и записи состояний, эти операции являются «состязательными». P4 требует атомарного (неделимого) выполнения вызовов методов и экземпляров extern. Для атомарного выполнения больших блоков кода в P4 применяется аннотация @atomic для блока операторов, состояния анализатора, блока управления или анализатора целиком. Рассмотрим пример.

```
extern Register { /* тело опущено */ }
control Ingress() {
        Register() r;
        table flowlet { /* чтение состояния r в действии (action) */ }
        table new_flowlet { /* запись состояния r в действии (action) */ }
        apply {
                @atomic {
                        flowlet.apply();
                        if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TIMEOUT)
                                new_flowlet.apply();
}}}
```

Программа обращается к внешнему объекту r типа Register в действиях из таблиц flowlet (чтение) и new_flowlet (запись). Без аннотации @atomic эти операции не будут выполняться атомарно и для второго пакета состояние r может быть считано до того, как первый изменит его.

Отметим, что даже в определении действия, которое может читать, изменять и записывать содержимое регистра, для блока работы с регистром следует использовать аннотацию @atomic, чтобы гарантировать следующему пакету доступ к обновлённой информации. Компилятор backend должен отвергать программы с блоками @atomic, если он не может реализовать неделимое выполнение последовательности инструкций. В таких случаях компилятору следует обеспечивать разумную диагностику.

18. Аннотации

Аннотации похожи на атрибуты C# и аннотации Java. Это просто механизм ограниченного расширения языка P4 без изменения грамматики. В какой-то степени аннотации включают #pragma из языка C. Аннотации присоединяются к типам, полям, переменным и т. п. с использованием синтаксиса @ (как показано явно в грамматике P4). Неструктурированные аннотации или просто аннотации могут не иметь тела, для структурированных тело обязательно и должно содержать хотя бы одну пару скобок [].

```
optAnnotations
        : /* пусто */
        | annotations
        ;
annotations
        : annotation
        | annotations annotation
        ;
annotation
        : '@' name
        | '@' name '(' annotationBody ')'
        | '@' name '[' structuredAnnotationBody ']'
        ;
```

Структурированные и неструктурированные аннотации для одного элемента не могут использовать одно имя. Т. е. данное имя для любого элемента может применяться лишь к одному из двух типов аннотаций. Аннотация, используемая для одного элемента, не влияет на аннотации других, поскольку у них своя область действия.

Действительные аннотации

```
@my_anno(1) table T { /* тело опущено */ }
@my_anno[2] table U { /* тело опущено */ } // разные области с предыдущей my_anno
```

Недействительные аннотации

```
@my_anno(1)
@my_anno[2] table U { /* тело опущено */ } // ошибка — изменён тип anno для элемента
```

Для данного элемента может задаваться множество неструктурированных аннотаций с одним именем - они аккумулируются и все будут связаны с этим элементом. Структурированная аннотация с данным именем для элемента может быть лишь одна и наличие нескольких аннотаций создаст ошибку.

Действительные аннотации

```
@my_anno(1)
@my_anno(2) table U { /* тело опущено */ }  // неструктурированные аннотации аккумулируются
```

Недействительные аннотации

```
@my_anno[1]
@my_anno[2] table U { /* тело опущено */ } // ошибка — та же структурированная аннотация
```

18.1. Тело неструктурированной аннотации

Гибкость неструктурированных аннотаций P4 обусловлена минимальной структурой, требуемой грамматикой P4 - тело неструктурированной аннотации может содержать любую последовательность терминалов при условии сбалансированности круглых скобок. В приведённом ниже фрагменте грамматики нетерминальный элемент annotationToken представляет любой терминал, созданный лексером, включая идентификаторы, ключевые слова, строки, и целочисленные литералы, но исключая круглые скобки.

```
annotationBody
        : /* пусто */
```

```
        | annotationBody '(' annotationBody ')'
        | annotationBody annotationToken
```

Неструктурированные аннотации могут иметь ту или иную структуру в своём теле, не определяемую языком P4. Например, спецификация P4Runtime определяет аннотацию @pkginfo, предполагающую пару ключ-значение.

18.2. Тело структурированных аннотаций

В отличие от неструктурированных аннотаций структурированные используют скобки [...] и формат их ограничен. Обычно такие аннотации служат для объявления пользовательских метаданных, состоящих из списков выражений или списков пар ключ-значение (но не обоих). Элемент expressionList может быть пустым или содержать список разделённых запятыми выражений. Элемент kvList включает одну или несколько пар kvPair, каждая из которых включает ключ и значение. Синтаксис выражения описан ниже (Приложение H. Грамматика P4).

Все выражения в structuredAnnotationBody должны иметь известные при компиляции значения - литералы или выражения, которые должны быть вычислены при компиляции, с типом результата string, int с неограниченной разрядностью или boolean. Структурированные выражения (например, выражения с expressionList, kvList и т. п.) не допускаются. Отметим, что информация P4Runtime (P4Info) может предусматривать дополнительные ограничения, например, целочисленные выражения могут быть ограничены 64-битовыми значениями. Не допускается дублирование ключей в kvList структурированной аннотации.

```
structuredAnnotationBody
        : expressionList
        | kvList
        ;
...
expressionList
        : /* пусто */
        | expression
        | expressionList ',' expression
        ;
...
kvList
        : kvPair
        | kvList ',' kvPair
        ;
kvPair
: name '=' expression
;
```

18.2.1. Примеры структурированных аннотаций

Пустой список выражений имеет пустую аннотацию

```
@Empty[]
table t { /* тело опущено */ }
```

Смешанный список выражения будет иметь аннотацию вида

```
[1,"hello",true, false, 11]
#define TEXT_CONST "hello"
#define NUM_CONST 6
@MixedExprList[1,TEXT_CONST,true,1==2,5+NUM_CONST]
table t { /* тело опущено */ }
```

Список строк kvList

```
@Labels[short="Short Label", hover="My Longer Table Label to appear in hover-help"]
table t { /* тело опущено */ }
```

Список смешанных выражений kvList имеет аннотацию

```
[label="text", my_bool=true, int_val=6]
@MixedKV[label="text", my_bool=true, int_val=2*3]
table t { /* тело опущено */ }
```

Список смешанных kvPair и expressionList будет недействительным, поскольку в нем смешаны kvPair и выражения

```
@IllegalMixing[key=4, 5] // недопустимое смешивание
table t { /* тело опущено */ }
```

Недействительное дублирование ключа

```
@DupKey[k1=4,k1=5] // недопустимое дублирование ключа
table t { /* тело опущено */ }
```

Недопустимое дублирование структурированных аннотаций

```
@DupAnno[k1=4]
@DupAnno[k2=5] // недопустимое дублирование имени
table t { /* тело опущено */ }
```

Недопустимое использование структурированной и неструктурированной аннотации

```
@MixAnno("Anything")
@MixAnno[k2=5] // недопустимое использование обоих типов аннотаций
table t { /* тело опущено */ }
```

18.3. Предопределённые аннотации

Имена аннотаций, начинающиеся со строчной буквы зарезервированы для стандартной библиотеки и архитектуры. Этот документ определяет «стандартные» аннотации в Приложении C. Предполагается, что этот список будет расти. Для архитектуры рекомендуется определять аннотации, начинающиеся с префикса производителя, например, организация X может использовать для аннотаций имена вида @X_annotation

18.3.1. Аннотации необязательных параметров

Параметр для пакета, внешнего метода, функции или объекта аннотируется с помощью @optional для указания того, что параметр не требует соответствующего аргумента. Значение параметра без аргумента зависит от платформы.

18.3.2. Аннотации списка действий таблицы

Для предоставления компилятору и плоскости управления дополнительной информации о действиях таблицы используются две аннотации, не имеющие тела:

- @tableonly указывает, что действие может присутствовать лишь в таблице и не применяется по умолчанию;
- @defaultonly указывает, что действие может применяться лишь по умолчанию, а не в таблице.

```
table t {
        actions = {
                a,              // может применяться везде
                @tableonly b,   // может применяться лишь в таблице
                @defaultonly c,         // может использоваться лишь по умолчанию
        }
        /* тело опущено */
}
```

18.3.3. Аннотации API плоскости управления

Аннотация @name указывает компилятору использовать другое локальное имя при генерации внешних API для манипуляций с объектом из плоскости управления. Телом аннотации является строковый литерал. В приведённом примере таблица имеет полное имя c_inst.t1.

```
control c( /* параметры опущены */ )() {
        @name("t1") table t { /* тело опущено */ }
        apply { /* тело опущено */ }
}
c() c_inst;
```

Аннотация @hidden скрывает управляемый элемент (например, таблицу, ключ, действие или extern) от плоскости управления, удаляя по сути полное имя (17.3. Имена элементов управления). Аннотация не имеет тела.

18.3.3.1. Ограничения

Для каждого элемента можно применять не более одной аннотации @name или @hidden, а каждое имя плоскости управления должно указывать не более одного управляемого объекта. Это вызывает озабоченность при использовании абсолютной аннотации @name - если тип, содержащий аннотацию @name с абсолютным путём (начинается с .) создаётся неоднократно, это приведёт к одному имени у нескольких управляемых элементов.

```
control noargs();
package top(noargs c1, noargs c2);
control c() {
        @name(".foo.bar") table t { /* тело опущено */ }
        apply { /* тело опущено */ }
}
top(c(), c()) main;
```

Без аннотации @name эта программа будет создавать два управляемых элемента с полными именами main.c1.t и main.c2.t. Однако аннотация @name(".foo.bar") переименуют в обоих экземплярах таблицу t в foo.bar и имена двух управляемых элементов совпадут,что недопустимо.

18.3.4. Аннотации одновременных элементов управления

Аннотация @atomic (17.4.1. Модель одновременной работы) позволяет обеспечить неделимость блока операций.

18.3.5. Аннотации наборов значений

Аннотация @match (12.6. Выражения для выбора) служит для задания match_kind, отличного принятого по умолчанию, для точного значения поля value_set.

18.3.6. Аннотации внешних функций и методов

В объявлениях внешних функций и методов могут появляться разные аннотации для описания ограничений в поведении и взаимодействиях этих функций. По умолчанию внешние функции могут оказывать любое влияние на среду программы P4 и могут взаимодействовать нетривиальными способами (6.7.1. Обоснование). Поскольку внешние объекты зависят от архитектуры и их поведение известно определению архитектуры, такие аннотации необязательны (реализация может знать о взаимодействии по их именам), но они обеспечивают единый способ описания некоторых общеизвестных взаимодействий (или их отсутствия) для независимого от архитектуры анализа программ P4.

- @pure описывает функцию, зависящую лишь от значений параметров и не оказывающую никакого влияния, кроме возврата значения и поведения copy-out для параметров out и inout. Между вызовами не записывается скрытых состояний и значение не зависит от скрытого состояния, которое могут менять другие вызовы. Примером является хэш-функция, которая рассчитывает детерминированную свёртку своих аргументов и возвращаемое значение не зависит от доступных для записи плоскости управления состояний или значения вектора инициализации. Функция @pure, результат которой не используется, может быть исключена без каких-либо неблагоприятных последствий, а множество вызовов с одними аргументами можно объединить в один вызов (с учётом ограничений поведения copy-out для параметров out и inout). Порядок вызова функций @pure можно менять относительно других расчётов, которые не зависят от данных.
- @noSideEffects слабее, чем @pure, и описывает функции, не меняющие скрытых состояний, но зависящие от таковых. Примером является хэш-функция, рассчитывающая детерминированную свёртку аргументов с учётом некого внутреннего состояния, которое может быть изменено через API плоскости управления (например,

вектор инициализации). Другим примером служит чтение одного элемента массива регистров объекта extern. Такая функция может быть исключена и можно изменить её порядок по отношению к другим вызовам @noSideEffects и @pure (с учётом ограничений поведения copy-out для параметров out и inout), но не к вызовам других функций, которые могут воздействовать на данную.

18.3.7. Аннотация отмены

Аннотация @deprecated должна включать строку аргумента с сообщением, выводимым компилятором при использовании в программе отменённой конструкции. Это полезно в библиотеках, объявляющих конструкции типа extern.

```
@deprecated("Please use the 'check' function instead")
extern Checker { /* тело опущено */ }
```

18.3.8. Отключение предупреждений

Аннотация noWarn имеет обязательный строковый аргумент, который указывает подавляемые предупреждения компилятора. Например @noWarn("unused") будет предотвращать вывод предупреждений компилятора о неиспользуемых объявлениях.

18.4. Зависимые от платформы аннотации

Каждая реализация компилятора P4 может определять свои аннотации с учётом целевой платформы. Синтаксис этих аннотаций должен соответствовать приведённым выше описаниям, семантика зависит от платформы. Аннотации можно применять аналогично pragma в других языках. Компилятору P4 следует выдавать:

- ошибку при некорректном использовании аннотаций (например, отсутствие или некорректный тип параметра);
- предупреждения при неизвестных аннотациях.

Приложение A. История выпусков

Версия	*Дата*	*Изменения*
1.0.0	17.05. 2017	Исходная версия.
1.1.0	26.11.2018	Добавлены функции верхнего уровня, необязательные и именованные параметры, представления enum, наборы значений анализатора, определения типов, арифметика с насыщением и структурированные аннотации. Исключена аннотация globalname, добавлено свойство таблицы size. Разъяснена семантика операций с недействительными заголовками, добавлены ограничения для аргументов вызова, изменён порядок выполнения побитовых операторов.
1.2.0	14.10.2019	Добавлена ошибка ParserInvalidArgument, порядок записей const, методы size для заголовков, 1-битовые значения со знаком, нарезки битов со знаком, пустые кортежи, аннотация @deprecated, аннотации в свободной форме, тип int для table.apply().miss, тип string.
1.2.1	11.06.2020	Добавлены выражения со значением struct, принятые по умолчанию значения, конкатенация, структурированные аннотации. Стандартизовано несколько новых аннотаций, обобщено правило типизации для масок, ограничено правило типизации для сдвига с операндами неограниченного размера. Разъяснён порядок оценки для ключей таблиц, поведение copy-out behavior, семантика недействительных стеков заголовков, семантика инициализации. Исправлено несколько мелких проблем в грамматике.
1.2.2	17.05.2021	Добавлена поддержка доступа к кортежам, универсальных (базовых) структур, дополнительных перечисляемых типов, абстрактных методов, условных и пустых операторов в синтаксических анализаторах, дополнительных приведений типов, типов с размером 0. Уточнена семантика заданных по умолчанию действий, заголовков, пустых типов и данных действий (action). Исправлены опечатки и несогласованности в грамматике.

A.1. Изменения в версии 1.2.2

- Добавлена поддержка доступа к полям кортежей (8.10. Операции над кортежами).
- Добавлена поддержка базовых структур (7.2.10. Специализация типа).
- Добавлена поддержка целочисленных значений, enum и error в операторах switch (11.7. Оператор выбора).
- Добавлена поддержка дополнительных перечисляемых типов (7.2.1. Перечисляемые типы).
- Добавлена поддержка абстрактных методов (7.2.9.2. Внешние объекты)[1].
- Добавлена поддержка условных и пустых операторов а синтаксических анализаторах (12.4. Состояния анализатора).
- Добавлена поддержка приведения int к bool (8.9.1. Явное приведение).
- Добавлена поддержка битовых строк и varbit нулевого размера (8.22. Чтение неинициализированных значений и запись полей в недействительные заголовки).
- Указано, что default_action имеет значение NoAction, если не указано иное (13.2. Таблицы).
- Указаны типы выражений, которые могут служить индексами в стеке заголовков (8.17. Операции над стеком заголовков).
- Разъяснено представление пустых типов (8.22. Чтение неинициализированных значений и запись полей в недействительные заголовки).

[1] *В оригинале ошибочно дана ссылка на раздел 7. Прим. перев.*

- Разъяснено, что данные действия могут быть заданы плоскостью управления, свойством таблицы dafault_entry и свойством таблицы const entries (13.1. Действия).
- Исправлены опечатки и несоответствия в грамматике (Приложение H. Грамматика P4).
- Исключены (обязательные) аннотации записей const в грамматике (Приложение H. Грамматика P4).

A.2. Изменения в версии 1.2.1

- Добавлены выражения со значением struct (8.12. Выражения со значением struct).
- Добавлена поддержка принятых по умолчанию значений (7.3. Подразумеваемые значения).
- Добавлена поддержка конкатенации строк (8.6.1. Конкатенация).
- Добавлены аннотации key-value и со структурой списка (18. Аннотации).
- Добавлены аннотации @pure и @noSideEffects (18.3.6. Аннотации внешних функций и методов).
- Добавлены аннотации @noWarn (18.3.8. Отключение предупреждений).
- Обобщена типизация масок для сериализации enum (18.3.3. Аннотации API плоскости управления).
- Ограничены положительными константами правые операнды битового сдвига с участием int (8.7. Операции над целыми числами произвольной точности).
- Разъяснено поведение copy-out для return (11.4. Оператор возврата) и exit (11.5. Оператор выхода).
- Разъяснена семантика недействительных стеков заголовков (8.17. Операции над стеком заголовков).
- Разъяснена семантика инициализации (6.6. Выражения для левой части, 6.7. Соглашения о вызовах), в частности, для заголовков и локальных переменных.
- Разъяснён порядок оценки ключей таблицы (13.2.3. Семантика выполнения блока СД).
- Уточнена грамматика для анализа сдвига вправо (>>) для поддержки пустых операторов в анализаторе (12.4. Состояния анализатора) и исключения аннотаций для записей const (13.2.1.4. Записи).
- Разъяснено представление логических значений в заголовках (7.2.2. Типы заголовков).

A.3. Изменения в версии 1.2.0

- Добавлено table.apply().miss (13.2.2. Вызов блока СД).
- Добавлен тип string (7.1.5. Строки).
- Добавлено неявное приведение для значений типа enum (8.3. Операции над типом enum).
- Добавлены 1-битовые значения со знаком.
- Нарезка битов из значений со знаком и без знака определена как беззнаковая.
- Ограничено местоположение метки default в операторах switch.
- Разрешены пустые кортежи.
- Добавлена аннотация @deprecated.
- Смягчены требования к структуре тела аннотаций.
- Исключена аннотация @pkginfo, перенесённая в спецификацию P4Runtime.
- Добавлен тип int (7.1.6.5. Целые числа “бесконечной точности”).
- Добавлена ошибка ParserInvalidArgument (12.8.2. Извлечение при переменном размере, 12.8.4. Пропуск битов).
- Разъяснена значимость порядка элементов в записях const (13.2.1.4. Записи).
- Добавлены методы расчёта размера заголовков (8.16. Операции над заголовками).

A.4. Изменения в версии 1.1.0

- Разрешено объявление функций на верхнем уровне программ P4 (9. Объявление функции).
- Разрешено задавать параметры по именам, с использованием принятого по умолчанию значения и делать параметры необязательными (6.7. Соглашения о вызовах)l.
- Добавлены перечисляемые значения - enum (8.3. Операции над типом enum).
- Добавлены наборы значений анализатора - value_set для программируемых плоскостью управление меток select (12.11. Набор значений анализатора).
- Разрешено определять новые типы в программах (7.5. Создание новых типов).
- Добавлена поддержка арифметики с насыщением для некоторых платформ (8.5. Операции над битовыми типами (целые числа без знака)).
- Добавлены структурированные аннотации как списки пар ключ-значение (18. Аннотации).
- Удалена аннотация globalname (17.3.2. Аннотации, управляющие именами).
- Добавлено необязательное свойство таблицы size (13.2.1.5. Размер).

- Разъяснена семантика операций с недействительными заголовками (8.16. Операции над заголовками).
- Добавлены ограничения на типы значений аргументов при вызовах (Приложение F. Ограничения для вызовов при компиляции и работе).
- Изменён порядок побитовых операторов (Приложение H. Грамматика P4) - &, | и ^ имеют более высокий приоритет, нежели <, >, <=, >=.
- Добавлена поддержка задания размера типов bit и varbit с использованием выражений (7.1. Базовые типы).

Приложение B. Зарезервированные слова P4

Ниже приведён список зарезервированных (ключевых) слов P4. Некоторые слова являются зарезервированными лишь в определённом контексте (например, actions).

action	apply	bit	bool	const	control	default	else	enum	error
extern	exit	false	header	header_union	if	in	inout	int	match_kind
package	parser	out	return	select	state	string	struct	switch	table
transition	true	tuple	typedef	varbit	verify	void			

Приложение C. Зарезервированные аннотации P4

Аннотация	*Назначение*	*Параграф*
atomic	Задаёт неделимое (атомарное) выполнение	17.4.1
defaultonly	Действие может использоваться лишь по умолчанию (default).	18.3.2
hidden	Скрывает управляемый объект от плоскости управления.	17.3.2
match	Задаёт поле match_kind в value_set.	18.3.5
name	Назначает локальное имя плоскости управления.	17.3.2
optional	Указывает необязательный параметр.	18.3.1
tableonly	Действие не может использоваться по умолчанию (default).	18.3.2
deprecated	Указывает отменённую (устаревшую) конструкцию.	18.3.7
pure	«Чистая» функция.	18.3.6
noSideEffects	Функция без побочных эффектов.	18.3.6
noWarn	Отключает предупреждения компилятора (аргумент типа string)	18.3.8

Приложение D. Основная библиотека P4

Основная библиотека P4 содержит объявления, нужные большинству программ. Например, она включает определения внешних объектов packet_in и packet_out, применяемых анализаторами и сборщиками при работе с пакетом.

```
/// Стандартные коды ошибок. Пользователи могут добавлять свои коды.
error {
        NoError,              /// Нет ошибок.
        PacketTooShort,       /// В пакете недостаточно битов для извлечения.
        NoMatch,              /// Выражение select не имеет совпадений.
        StackOutOfBounds,     /// ссылка на недействительный элемент стека заголовков.
        HeaderTooShort,       /// Извлечение излишнего числа битов в поле varbit.
        ParserTimeout, /// Превышено время работы анализатора.
        ParserInvalidArgument /// Операция анализатора вызвана с неподдерживаемым
                                  /// реализацией значением.
}
extern packet_in {
        /// Чтение заголовка из пакета в заголовок фиксированного размера @hdr
        /// и перемещение указателя. Может вызывать ошибку PacketTooShort или
        /// StackOutOfBounds. @T - тип заголовка с фиксированным размером.
        void extract<T>(out T hdr);
        /// Чтение битов из пакета в заголовок переменного размера @variableSizeHeader
        /// и перемещение указателя. Заголовок @T должен содержать 1 поле varbit.
        /// Может вызывать ошибки PacketTooShort, StackOutOfBounds, HeaderTooShort.
        void extract<T>(out T variableSizeHeader,
                in bit<32> variableFieldSizeInBits);
        /// Чтение битов из пакета без перемещения указателя. @returns содержит
        /// прочитанные биты T может быть произвольным фиксированным заголовком.
        T lookahead<T>();
        /// Перемещение указателя на заданное число битов.
        void advance(in bit<32> sizeInBits);
        /// @return - размер пакета в байтах. Метод поддерживается не всеми
        /// архитектурами.
        bit<32> length();
}
extern packet_out {
        /// запись @data в выходной пакет с пропуском недействительных заголовков
        /// и перемещением указателя. @T может быть заголовком, стеком или
        /// объединением заголовков, а также struct с полями этих типов.
        void emit<T>(in T data);
}
action NoAction() {}
/// Стандартные типы сопоставления для полей ключей в таблице. Некоторые
/// архитектуры могут поддерживать не все типы сопоставления. Архитектура
/// может задавать свои типы.
match_kind {
        exact,          /// Точное сопоставление.
        ternary,        /// Троичное сопоставление с использованием маски.
        lpm             /// Наибольший совпадающий префикс.
}
```

Приложение E. Контрольные суммы

В $P4_{16}$ нет встроенных конструкций для работы с контрольными суммами. Предполагается выполнение таких операций внешними объектами, обеспечиваемыми зависимыми от платформы библиотеками. Библиотеке стандартной архитектуры следует включать блоки работы с контрольными суммами. Например, можно предоставить блок инкрементного расчёта контрольных сумм Checksum16 (5.2.4. Доступные внешние блоки) для 16-битовых контрольных сумм с дополнением до 1 с использованием внешнего объекта, как показано ниже.

```
extern Checksum16 {
       Checksum16();          // конструктор
       void clear();          // подготовка блока к расчётам
       void update<T>(in T data);    // добавление данных в контрольную сумму
       void remove<T>(in T data);    // исключение данных из контрольной суммы
       bit<16> get();         // расчёт контрольной суммы для данных, добавленных после очистки
}
```

Проверку контрольной суммы IP можно организовать в коде анализатора, как показано ниже.

```
ck16.clear();          // подготовка блока контрольной суммы
ck16.update(h.ipv4);   // запись заголовка
verify(ck16.get() == 16w0, error.IPv4ChecksumError); // проверка значения 0
```

Расчёт контрольной суммы IP можно выполнить в форме

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

В некоторых вариантах архитектуры коммутаторов контрольные суммы не проверяются, а лишь обновляются с учётом внесённых изменений. Это можно реализовать в программе P4, как показано ниже.

```
ck16.clear();
ck16.update(h.ipv4.hdrChecksum);      // исходная контрольная сумма
ck16.remove( { h.ipv4.ttl, h.ipv4.proto } );
h.ipv4.ttl = h.ipv4.ttl - 1;
ck16.update( { h.ipv4.ttl, h.ipv4.proto } );
h.ipv4.hdrChecksum = ck16.get();
```

Приложение F. Ограничения для вызовов при компиляции и работе

В этом приложении описаны ограничения при компиляции и работе программ., многие из которых уже упоминались.

Объектами с состоянием в $P4_{16}$ являются пакеты (package), анализаторы, элементы управления, внешние объекты (extern), таблицы и наборы значений. Функции $P4_{16}$ также относятся к этой группе, даже если они зависят лишь от своих аргументов (pure). Все остальные типы здесь называются типами значений (value type).

Некоторые базовые принципы перечислены ниже.

- Элементам управления не разрешено вызывать анализаторы и наоборот, поэтому нет смысла передавать один тип другому в параметрах конструктора или параметрах среды выполнения.
- При работе после вызова элемента управления, но до его завершения не может быть рекурсивных вызовов между элементами управления или к самому себе. Аналогичное условие относится и к анализаторам, поскольку возможны зацикливания состояния внутри анализатора.
- Внешним элементам не разрешено вызывать анализаторы и элементы управления, поэтому нет смысла передавать им такие объекты.
- Экземпляры таблиц всегда создаются напрямую во включающих таблицы элементах управления и не могут создаваться на верхнем уровне. Синтаксис задания параметров таблиц не определяется. Таблица предназначена для использования лишь в определившем её элементе управления.
- Наборы значений создаются во включающем их анализаторе или на верхнем уровне. Синтаксис задания параметров набора значений не определяется. Наборы значений могут использоваться совместно несколькими анализаторами, если они находятся в области действия.

Предполагается, что некоторые архитектуры будут поддерживать рециркуляцию пакетов, когда обработанный пакет или его «клон» снова попадает на вход. Это не отменяет приведённых ранее замечаний о рекурсии, относящихся к работе анализаторов и элементов управления.

В таблице показано, какие типы могут передаваться в качестве параметров конструктора для другого типа

	Может быть параметром конструктора для типа			
Тип	***package***	***parser***	***control***	***extern***
package	да	нет	нет	нет
parser	да	да	нет	нет
control	да	нет	да	нет
extern	да	да	да	да
function	нет	нет	нет	нет
table	нет	нет	нет	нет
value-set	нет	нет	нет	нет
Типы значений	да	да	да	да

В следующей таблице показаны ограничения на создание экземпляров (10.3. Создание экземпляров) разных типов. Ответы «нет» в столбце означают, что нет создания экземпляров «внутри пакета» в $P4_{16}$. Можно явно вызвать конструктор и использовать экземпляры типов с состояниями при создании экземпляра пакета с учётом приведённых ниже ограничений. Для типов extern можно указать лишь интерфейс в $P4_{16}$, но не реализацию, поэтому экземпляры внутри extern не создаются. Можно объявить переменные и константы любого из типов значений внутри анализатора,

элемента управления или функции (10.2. Переменные). Такое объявление отличается от создания экземпляра, поэтому с таблице указано «-». Переменные нельзя объявлять на верхнем уровне программы, а константы можно.

		Экземпляр может создаваться в				
Тип	***Верхний уровень***	***package***	***parser***	***control***	***extern***	***function***
package	да	нет	нет	нет	нет	нет
parser	нет	нет	да	нет	нет	нет
control	нет	нет	нет	да	нет	нет
extern	да	нет	да	да	нет	нет
function	да	нет	нет	нет	нет	нет
table	нет	нет	нет	да	нет	нет
value-set	да	нет	да	нет	нет	нет
Типы значений	-	-	-	-	-	-

В следующей таблице приведены ограничения для типов, которые могут передаваться в качестве параметров при работе другим элементам, способным принимать такие параметры (parser, control, extern, action, function).

	Может быть параметром в среде выполнения для вызова				
Тип	***parser***	***control***	***method***	***action***	***function***
package	нет	нет	нет	нет	нет
parser	нет	нет	нет	нет	нет
control	нет	нет	нет	нет	нет
extern	да	да	да	нет	нет
table	нет	нет	нет	нет	нет
value-set	нет	нет	нет	нет	нет
action	нет	нет	нет	нет	нет
function	нет	нет	нет	нет	нет
Типы значений	да	да	да	да	да

Вызовы метода extern могут возвращать лишь типы значений или не возвращать ничего (return void).

В следующей таблице показаны ограничения для вызовов, доступных из разных мест программы P4. Вызов анализатора, элемента управления или таблицы означает применение метода apply(), вызов value-set - использование в выражении select. Строка для extern указывает, откуда можно вызывать внешние методы. Одним из способов вызова extern с верхнего уровня анализатора или элемента управления является выражение инициализатора для объявленной переменной, например, bit<32> x = rand.get();.

	Может вызываться при работе из указанного места программы P4					
Тип	***Состояние анализатора***	***Метод apply элемента управления***	***Верхний уровень анализатора или элемента управления***	***action***	***extern***	***function***
package	-	-	-	-	-	-
parser	да	нет	нет	нет	нет	нет
control	нет	да	нет	нет	нет	нет
extern	да	да	да	да	нет	нет
table	нет	да	нет	нет	нет	нет
value-set	да	нет	нет	нет	нет	нет
action	нет	да	нет	да	нет	нет
function	да	да	нет	да	нет	да
Типы значений	-	-	-	-	-	-

Вызовы не допускают рекурсии ни напрямую, ни опосредованно (взаимная рекурсия). Методы extern не могут вызывать другие типы программных объектов P4 (6.7.1. Обоснование). Действия могут вызываться напрямую из блока apply элементов управления.

Хотя строка extern показывает возможность вызова внешних методов из разных мест, конкретные методы могут вносить дополнительные ограничения. Такие ограничения следует описывать в документации метода для архитектуры, определяющей его. Во многих случаях ограничения могут относится лишь к состояниями анализатора или действиям блока управления, например, возможность вызова лишь из конкретного элемента управления, созданного в архитектуре для определённой роли.

Приложение G. Нерешенные проблемы

Сейчас остаётся ещё много открытых вопросов, обсуждаемых в рабочей группе P4 и кратко рассмотренных здесь. Разработчики приглашают сообщество принимать участие в обсуждении этих вопросов и экспериментировать с разными реализациями компиляторов.

G.1. Обобщённое поведение оператора switch

$P4_{16}$ включает операторы switch (11.7. Оператор выбора) и выражения select (12.6. Выражения для выбора). Они реально различаются в текущей версии языка. Оператор должен оцениваться в значение состояния. Предлагается обобщенный оператор switch, соответствующий принятым в большинстве языков программирования - условие со множеством вариантов, из которых выбирается первый подходящий.

```
switch(e1,/* параметры опущены */,en) {
        pat_1 : stmt1;
        /* тело опущено */
        pat_m : stmtm;
}
```

В примере проверяемое значение задано кортежем (e1,/* параметры опущены */,en), а варианты - выражениями, обозначающими наборы значений. Значение соответствует варианту (branch), если оно входит в набор, заданный выражением. В отличие от C и C++, здесь не применяется оператор break для предотвращения перехода к следующему варианту и такой переход возможен лишь при несоответствии значения текущей ветви (метке).

Это сделано для фиксации стандартной семантики операторов switch и поддержки общей идиомы анализаторов P4, где операторы применяются для смены состояний анализатора в зависимости от одного или нескольких значений, проанализированных ранее. Используя оператор switch, можно также обобщить устройство анализаторов, исключив select и сняв большинство ограничений на тип операторов, которые могут присутствовать в состоянии. В частности, разрешаются условные операторы и select с произвольной вложенностью. Этот язык можно перенести в более ограниченную версию, где тело каждого состояния содержит последовательность объявлений переменных, присваиваний и вызовов методов, за которыми следует одиночный оператор смены состояния.

Обобщено также устройство обработки для таблиц совпадений и отсутствия (hit/miss) и действий в блоках управления, путём генерации неявных типов для действий и результатов.

Контраргументом для этого предложения является то, что семантика select в анализаторе достаточно сильно отличается от оператора switch.

G.2. Неопределённое поведение

Неопределённость поведения вызывает множество проблем в таких языках, как C и HTML, включая ошибки и серьёзные уязвимости защиты. Есть несколько мест, где оценка программы P4 может приводить к неопределённому поведению - параметры out, неинициализированные переменные, доступ к полям недействительных заголовков или к стекам заголовков за пределами границы стека. Нужно сделать все возможное для устранения неопределённостей, в $P4_{16}$, поэтому предлагается усиливать формулировки спецификации, чтобы исключить описанное выше поведение в принятых по умолчанию случаях. С учётом заботы о производительности предлагается определить флаги компиляции и/или pragma для переопределения заданного по умолчанию поведения. Однако предполагается, что программистам следует создавать защищённые программы и более серьёзно относиться к вопросам безопасности.

G.3. Структурированные итерации

Введение итератора в стиле foreach для работы со стеками заголовков снижает необходимость использования директив препроцессора C для определения размера стеков заголовков. Например,

```
foreach hdr in hdrs { /// операции над HDR }
```

Поскольку стеки всегда известны статически (при компиляции), компилятор может преобразовать оператор foreach в реплицированный код с явными ссылками на индексы во время компиляции. Преимуществом этого является возможность писать код без учёта размера параметризованного стека заголовков. Поскольку компилятор может статически определять число операций, которые будут получены из foreach, он может отвергать программу, если она требует для действий слишком много ресурсов, или разделять код действия в соответствии с доступными ресурсами.

Приложение H. Грамматика P4

Грамматика $P4_{16}$ описана на языке YACC/bison и не задаёт приоритет операций. Грамматика в реальности неоднозначна, поэтому лексер и синтаксический анализатор должны работать совместно. В частности, лексер должен различать ранее введённые идентификаторы типов (маркеры TYPE_IDENTIFIER) и обычные идентификаторы (маркер IDENTIFIER). Анализатор должен использовать таблицу символов, чтобы указывать лексеру, как следует разбирать последовательные идентификаторы. Например, для приведённого ниже фрагмента

```
typedef bit<4> t;
struct s { /* тело опущено */}
t x;
parser p(bit<8> b) { /* тело опущено */ }
```

лексер возвратит показанные ниже виды терминалов

t - TYPE_IDENTIFIER
s - TYPE_IDENTIFIER
x - IDENTIFIER
p - TYPE_IDENTIFIER
b - IDENTIFIER

На эту грамматику оказывают существенное влияние ограничения инструмента генерации анализаторов Bison.

Несколько других терминалов констант присутствуют в приведённых ниже правилах

MASK - это &&&
RANGE - это ..
DONTCARE - это _

Маркер STRING_LITERAL соответствует строковому литералу в двойных кавычках (6.3.3.3. Строковые литералы).

Все остальные терминалы являются представлением ключевых слов заглавными буквами. Например, RETURN - это терминал, возвращаемый лексером при анализе ключевого слова return.

```
p4program
        : /* пусто */
        | p4program declaration
        | p4program ';'       /* пустое объявление */
        ;

declaration
        : constantDeclaration
        | externDeclaration
        | actionDeclaration
        | parserDeclaration
        | typeDeclaration
        | controlDeclaration
        | instantiation
```

```
        | errorDeclaration
        | matchKindDeclaration
        | functionDeclaration
        ;

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

name
        : nonTypeName
        | TYPE_IDENTIFIER
        ;

nonTableKwName
        : IDENTIFIER
        | TYPE_IDENTIFIER
        | APPLY
        | STATE
        | TYPE
        ;

optAnnotations
        : /* пусто */
        | annotations
        ;

annotations
        : annotation
        | annotations annotation
        ;

annotation
        : '@' name
        | '@' name '(' annotationBody ')'
        | '@' name '[' structuredAnnotationBody ']'
        ;

parameterList
        : /* пусто */
        | nonEmptyParameterList
        ;

nonEmptyParameterList
        : parameter
        | nonEmptyParameterList ',' parameter
        ;

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

direction
        : IN
        | OUT
        | INOUT
        : /* пусто */
        ;

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

instantiation
    : typeRef '(' argumentList ')' name ';'
    | annotations typeRef '(' argumentList ')' name ';'
    | annotations typeRef '(' argumentList ')' name '=' objInitializer ';'
    | typeRef '(' argumentList ')' name '=' objInitializer ';'
    ;

objInitializer
    : '{' objDeclarations '}'
    ;

objDeclarations
    : /* пусто */
    | objDeclarations objDeclaration
```

```
    ;

objDeclaration
    : functionDeclaration
    | instantiation
    ;

optConstructorParameters
        : /* пусто */
        | '(' parameterList ')'
        ;

dotPrefix
        : '.'
        ;

/************************** Анализатор ****************************/
parserDeclaration
        : parserTypeDeclaration optConstructorParameters
          /* параметры типа не разрешены в parserTypeDeclaration */
          '{' parserLocalElements parserStates '}'
        ;

parserLocalElements
        : /* пусто */
        | parserLocalElements parserLocalElement
        ;

parserLocalElement
        : constantDeclaration
        | variableDeclaration
        | instantiation
        | valueSetDeclaration
        ;

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

parserStates
        : parserState
        | parserStates parserState
        ;

parserState
        : optAnnotations STATE name '{' parserStatements transitionStatement '}'
        ;

parserStatements
        : /* пусто */
        | parserStatements parserStatement
        ;

parserStatement
        : assignmentOrMethodCallStatement
        | directApplication
        | parserBlockStatement
        | constantDeclaration
        | variableDeclaration
        | emptyStatement
        | conditionalStatement
        ;

parserBlockStatement
        : optAnnotations '{' parserStatements '}'
        ;

transitionStatement
        : /* пусто */
        | TRANSITION stateExpression
        ;

stateExpression
        : name ';'
        | selectExpression
        ;

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

selectCaseList
        : /* пусто */
        | selectCaseList selectCase
        ;
```

```
selectCase
        : keysetExpression ':' name ';'
        ;

keysetExpression
        : tupleKeysetExpression
        | simpleKeysetExpression
        ;

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

simpleExpressionList
        : simpleKeysetExpression
        | simpleExpressionList ',' simpleKeysetExpression
        ;

simpleKeysetExpression
        : expression
        | DEFAULT
        | DONTCARE
        | expression MASK expression
        | expression RANGE expression
        ;

valueSetDeclaration
        : optAnnotations
          VALUESET '<' baseType '>' '(' expression ')' name ';'
        | optAnnotations
          VALUESET '<' tupleType '>' '(' expression ')' name ';'
        | optAnnotations
          VALUESET '<' typeName '>' '(' expression ')' name ';'
        ;

/************************** Элемент управления ***********************/
controlDeclaration
        : controlTypeDeclaration optConstructorParameters
          /* параметры типа не разрешены в controlTypeDeclaration */
          '{' controlLocalDeclarations APPLY controlBody '}'
        ;

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

controlLocalDeclarations
        : /* пусто */
        | controlLocalDeclarations controlLocalDeclaration
        ;

controlLocalDeclaration
        : constantDeclaration
        | actionDeclaration
        | tableDeclaration
        | instantiation
        | variableDeclaration
        ;

controlBody
        : blockStatement
        ;

/************************** Внешний объект ***********************/
externDeclaration
        : optAnnotations EXTERN nonTypeName optTypeParameters '{' methodPrototypes '}'
        | optAnnotations EXTERN functionPrototype ';'
        ;

methodPrototypes
        : /* пусто */
        | methodPrototypes methodPrototype
        ;

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

methodPrototype
        : optAnnotations functionPrototype ';'
        | optAnnotations TYPE_IDENTIFIER '(' parameterList ')' ';'
        ;
```

```
/************************* Типы ****************************/
typeRef
        : baseType
        | typeName
        | specializedType
        | headerStackType
        | tupleType
        ;

namedType
        : typeName
        | specializedType
        ;

prefixedType
        : TYPE_IDENTIFIER
        | dotPrefix TYPE_IDENTIFIER
        ;

typeName
        : prefixedType
        ;

tupleType
        : TUPLE '<' typeArgumentList '>'
        ;

headerStackType
        : typeName '[' expression ']'
        | specializedType '[' expression ']'
        ;

specializedType
        : prefixedType '<' typeArgumentList '>'
        ;

baseType
        : BOOL
        | ERROR
        | INT
        | BIT
        | BIT '<' INTEGER '>'
        | INT '<' INTEGER '>'
        | VARBIT '<' INTEGER '>'
        | BIT '<' '(' expression ')' '>'
        | INT '<' '(' expression ')' '>'
        | VARBIT '<' '(' expression ')' '>'
        ;

typeOrVoid
        : typeRef
        | VOID
        | IDENTIFIER   // может быть переменной типа
        ;

optTypeParameters
        : /* пусто */
        | '<' typeParameterList '>'
        ;

typeParameterList
        : name
        | typeParameterList ',' name
        ;
realTypeArg
        : DONTCARE
        | typeRef
        | VOID
        ;

typeArg
        : DONTCARE
        | typeRef
        | nonTypeName
        | VOID
        ;

realTypeArgumentList
        : realTypeArg
        | realTypeArgumentList COMMA typeArg
        ;

typeArgumentList
        : /* пусто */
```

```
        | typeArg
        | typeArgumentList ',' typeArg
        ;

typeDeclaration
        : derivedTypeDeclaration
        | typedefDeclaration
        | parserTypeDeclaration ';'
        | controlTypeDeclaration ';'
        | packageTypeDeclaration ';'
        ;

derivedTypeDeclaration
        : headerTypeDeclaration
        | headerUnionDeclaration
        | structTypeDeclaration
        | enumDeclaration
        ;

headerTypeDeclaration
        : optAnnotations HEADER name '{' structFieldList '}'
        ;

headerUnionDeclaration
        : optAnnotations HEADER_UNION name '{' structFieldList '}'
        ;

structTypeDeclaration
        : optAnnotations STRUCT name '{' structFieldList '}'
        ;

structFieldList
        : /* пусто */
        | structFieldList structField
        ;

structField
        : optAnnotations typeRef name ';'
        ;

enumDeclaration
        : optAnnotations ENUM name '{' identifierList '}'
        | optAnnotations ENUM BIT '<' INTEGER '>' name '{' specifiedIdentifierList '}'
        ;

errorDeclaration
        : ERROR '{' identifierList '}'
        ;

matchKindDeclaration
        : MATCH_KIND '{' identifierList '}'
        ;

identifierList
        : name
        | identifierList ',' name
        ;

specifiedIdentifierList
        : specifiedIdentifier
        | specifiedIdentifierList ',' specifiedIdentifier
        ;

specifiedIdentifier
        : name '=' initializer
        ;

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

/************************** Операторы ************************/
assignmentOrMethodCallStatement
        : lvalue '(' argumentList ')' ';'
        | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
        | lvalue '=' expression ';'
        ;

emptyStatement
        : ';'
        ;

returnStatement
```

```
        : RETURN ';'
        | RETURN expression ';'
        ;

exitStatement
        : EXIT ';'
        ;

conditionalStatement
        : IF '(' expression ')' statement
        | IF '(' expression ')' statement ELSE statement
        ;

// для поддержки прямых вызовов элемента управления или анализатора
directApplication
        : typeName '.' APPLY '(' argumentList ')' ';'
        ;

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

blockStatement
        : optAnnotations '{' statOrDeclList '}'
        ;

statOrDeclList
        : /* пусто */
        | statOrDeclList statementOrDeclaration
        ;

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

switchCases
        : /* пусто */
        | switchCases switchCase
        ;

switchCase
        : switchLabel ':' blockStatement
        | switchLabel ':'
        ;

switchLabel
        : name
        | DEFAULT
        | nonBraceExpression
        ;

statementOrDeclaration
        : variableDeclaration
        | constantDeclaration
        | statement
        | instantiation
        ;

/************ Таблицы *************/
tableDeclaration
        : optAnnotations TABLE name '{' tablePropertyList '}'
        ;

tablePropertyList
        : tableProperty
        | tablePropertyList tableProperty
        ;

tableProperty
        : KEY '=' '{' keyElementList '}'
        | ACTIONS '=' '{' actionList '}'
        | optAnnotations CONST ENTRIES '=' '{' entriesList '}' /* неизменяемые записи */
        | optAnnotations CONST nonTableKwName '=' initializer ';'
        | optAnnotations nonTableKwName '=' initializer ';'
        ;

keyElementList
        : /* пусто */
        | keyElementList keyElement
```

```
        ;

keyElement
        : expression ':' name optAnnotations ';'
        ;

actionList
        : /* пусто */
        | actionList optAnnotations actionRef ';'
        ;

actionRef
        : prefixedNonTypeName
        | prefixedNonTypeName '(' argumentList ')'
        ;

entriesList
        : entry
        | entriesList entry
        ;

entry
        : keysetExpression ':' actionRef optAnnotations ';'
        ;

/************************ Действия ******************************/
actionDeclaration
        : optAnnotations ACTION name '(' parameterList ')' blockStatement
        ;

/************************ Переменные ****************************/
variableDeclaration
        : annotations typeRef name optInitializer ';'
        | typeRef name optInitializer ';'
        ;

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

optInitializer
        : /* пусто */
        | '=' initializer
        ;

initializer
        : expression
        ;

/************************ Выражения *****************************/
functionDeclaration
        : functionPrototype blockStatement
        ;

argumentList
        : /* пусто */
        | nonEmptyArgList
        ;

nonEmptyArgList
        : argument
        | nonEmptyArgList ',' argument
        ;

argument
        : expression
        | name '=' expression
        | DONTCARE
        ;

kvList
        : kvPair
        | kvList ',' kvPair
        ;

kvPair
        : name '=' expression
        ;

expressionList
        : /* пусто */
        | expression
        | expressionList ',' expression
        ;

annotationBody
```

```
        : /* пусто */
        | annotationBody '(' annotationBody ')'
        | annotationBody annotationToken
        ;

structuredAnnotationBody
        : expressionList
        | kvList
        ;

annotationToken
        : ABSTRACT
        | ACTION
        | ACTIONS
        | APPLY
        | BOOL
        | BIT
        | CONST
        | CONTROL
        | DEFAULT
        | ELSE
        | ENTRIES
        | ENUM
        | ERROR
        | EXIT
        | EXTERN
        | FALSE
        | HEADER
        | HEADER_UNION
        | IF
        | IN
        | INOUT
        | INT
        | KEY
        | MATCH_KIND
        | OUT
        | PARSER
        | PACKAGE
        | PRAGMA
        | RETURN
        | SELECT
        | STATE
        | STRING
        | STRUCT
        | SWITCH
        | TABLE
        | THIS
        | TRANSITION
        | TRUE
        | TUPLE
        | TYPE
        | TYPEDEF
        | VARBIT
        | VALUESET
        | VOID
        | "_"
        | IDENTIFIER
        | TYPE_IDENTIFIER
        | STRING_LITERAL
        | INTEGER
        | "&&&"
        | ".."
        | "<<"
        | "&&"
        | "||"
        | "=="
        | "!="
        | ">="
        | "<="
        | "++"
        | "+"
        | "|+|"
        | "-"
        | "|-|"
        | "*"
        | "/"
        | "%"
        | "|"
        | "&"
        | "^"
        | "~"
        | "["
        | "]"
        | "{"
        | "}"
        | "<"
```

```
        | ">"
        | "!"
        | ":"
        | ","
        | "?"
        | "."
        | "="
        | ";"
        | "@"
        | UNKNOWN_TOKEN

member
        : name
        ;

prefixedNonTypeName
        : nonTypeName
        | dotPrefix nonTypeName
        ;

lvalue
        : prefixedNonTypeName
        | THIS
        | lvalue '.' member
        | lvalue '[' expression ']'
        | lvalue '[' expression ':' expression ']'
        ;

%left ','
%nonassoc '?'
%nonassoc ':'
%left '||'
%left '&&'
%left '==' '!='
%left '<' '>' '<=' '>='
%left '|'
%left '^'
%left '&'
%left '<<' '>>'
%left '++' '+' '-' '|+|' '|-|'
%left '*' '/' '%'
%right PREFIX
%nonassoc ']' '(' '['
%left '.'

// Дополнительные предпочтения, которые нужно задать
expression
        : INTEGER
        | TRUE
        | FALSE
        | STRING_LITERAL
        | nonTypeName
        | dotPrefix nonTypeName
        | expression '[' expression ']'
        | expression '[' expression ':' expression ']'
        | '{' expressionList '}'
        | '{' kvList '}'
        | '(' expression ')'
        | '!' expression %prec PREFIX
        | '~' expression %prec PREFIX
        | '-' expression %prec PREFIX
        | '+' expression %prec PREFIX
        | typeName '.' member
        | ERROR '.' member
        | expression '.' member
        | expression '*' expression
        | expression '/' expression
        | expression '%' expression
        | expression '+' expression
        | expression '-' expression
        | expression '|+|' expression
        | expression '|-|' expression
        | expression '<<' expression
        | expression '>>' expression
        | expression '<=' expression
        | expression '>=' expression
        | expression '<' expression
        | expression '>' expression
        | expression '!=' expression
        | expression '==' expression
        | expression '&' expression
        | expression '^' expression
        | expression '|' expression
        | expression '++' expression
        | expression '&&' expression
        | expression '||' expression
```

```
        | expression '?' expression ':' expression
        | expression '<' realTypeArgumentList '>' '(' argumentList ')'
        | expression '(' argumentList ')'
        | namedType '(' argumentList ')'
        | '(' typeRef ')' expression
        ;
nonBraceExpression
        : INTEGER
        | STRING_LITERAL
        | TRUE
        | FALSE
        | THIS
        | nonTypeName
        | dotPrefix nonTypeName
        | nonBraceExpression '[' expression ']'
        | nonBraceExpression '[' expression ':' expression ']'
        | '(' expression ')'
        | '!' expression %prec PREFIX
        | '~' expression %prec PREFIX
        | '-' expression %prec PREFIX
        | '+' expression %prec PREFIX
        | typeName '.' member
        | ERROR '.' member
        | nonBraceExpression '.' member
        | nonBraceExpression '*' expression
        | nonBraceExpression '/' expression
        | nonBraceExpression '%' expression
        | nonBraceExpression '+' expression
        | nonBraceExpression '-' expression
        | nonBraceExpression '|+|' expression
        | nonBraceExpression '|-|' expression
        | nonBraceExpression '<<' expression
        | nonBraceExpression '>>' expression
        | nonBraceExpression '<=' expression
        | nonBraceExpression '>=' expression
        | nonBraceExpression '<' expression
        | nonBraceExpression '>' expression
        | nonBraceExpression '!=' expression
        | nonBraceExpression '==' expression
        | nonBraceExpression '&' expression
        | nonBraceExpression '^' expression
        | nonBraceExpression '|' expression
        | nonBraceExpression '++' expression
        | nonBraceExpression '&&' expression
        | nonBraceExpression '||' expression
        | nonBraceExpression '?' expression ':' expression
        | nonBraceExpression '<' realTypeArgumentList '>' '(' argumentList ')'
        | nonBraceExpression '(' argumentList ')'
        | namedType '(' argumentList ')'
        | '(' typeRef ')' expression
        ;
```

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru