Спецификация языка версии $P4_{16}$ 1.0.0

$P4_{16}$ Language Specification

version 1.0.0

The P4 Language Consortium

2017-05-22

Тезисы

P4 является языком для программирования уровня данных (data plane) сетевых устройств. В этом документе приведено точное определение языка $P4_{16}$, который является пересмотренным в 2016 году вариантом языка P4 (http://p4.org).

Этот документ адресован разработчикам, создающим компиляторы, модели, среды разработки (IDE) и отладчики для программ P4. Документ может также заинтересовать программистов, которые хотят глубже разобраться в синтаксисе и семантике языка P4.

Оглавление

1. Назначение языка....4
2. Термины, определения и символы....4
3. Обзор....4
  3.1. Преимущества P4....6
  3.2. Развитие языка P4 - сравнение с предыдущими версиями (P4 v1.0/v1.1)....6
4. Модель архитектуры....7
  4.1. Стандартные модели архитектуры....8
  4.2. Интерфейсы уровня данных....8
  4.3. Внешние объекты и функции....8
5. Пример очень простого коммутатора....8
  5.1. Архитектура очень простого коммутатора....9
  5.2. Описание архитектуры VSS....10
    5.2.1. Блок арбитража....10
    5.2.2. Блок анализатора....11
    5.2.3. Блок демультиплексирования....11
    5.2.4. Доступные внешние блоки....11
  5.3. Полная программа VSS....12
6. Определение языка P4....14
  6.1. Синтаксис и семантика....15
    6.1.1. Грамматика....15
    6.1.2. Семантика и абстрактные машины P4....15
  6.2. Предварительная обработка....15
    6.2.1. Библиотека ядра P4....15
  6.3. Лексические конструкции....15
    6.3.1. Идентификаторы....16
    6.3.2. Комментарии....16
    6.3.3. Константы....16
      6.3.3.1. Логические константы....16
      6.3.3.2. Целочисленные константы....16
      6.3.3.3. Строковые литералы....17
  6.4. Соглашения об именовании....17
  6.5. Программы P4....17
    6.5.1. Области действия....17
    6.5.2. Элементы с внутренними состояниями....18
  6.6. L-значения....18
  6.7. Соглашения о вызовах - copy in/copy out....18
    6.7.1. Обоснование....19
  6.8. Преобразование имён....20
  6.9. Видимость....20
7. Типы данных P4....20
  7.1. Базовые типы....20
    7.1.1. Тип void....21
    7.1.2. Тип error....21
    7.1.3. Тип match_kind....21
    7.1.4. Тип bool....21
    7.1.5. Строки....21
    7.1.6. Целые числа....21
      7.1.6.1. Переносимость....22
      7.1.6.2. Целые числа без знака (битовые строки)....22
      7.1.6.3. Целые числа со знаком....22
      7.1.6.4. Битовые строки с динамическим размером....22

7.1.6.5. Целые числа с неограниченной разрядностью....22
7.1.6.6. Типы целочисленных констант....22
7.2. Производные типы....23
7.2.1. Перечисляемые типы....23
7.2.2. Типы заголовков....24
7.2.3. Стеки заголовков....24
7.2.4. Объединения заголовков....25
7.2.5. Структурированные типы....25
7.2.6. Кортежи....25
7.2.7. Правила для вложенных типов....25
7.2.8. Синтезированные типы данных....26
7.2.8.1. Типы set....26
7.2.8.2. Типы function....26
7.2.9. Внешние типы....26
7.2.9.1. Внешние функции....26
7.2.9.2. Внешние объекты....26
7.2.10. Специализация типа....27
7.2.11. Типы анализаторов и управляющих блоков....27
7.2.11.1. Объявление типа анализатора....27
7.2.11.2. Объявления типа элемента управления....27
7.2.12. Типы программ....27
7.2.13. Типы _....27
7.3. typedef....27
8. Выражения....28
8.1. Порядок вычисления выражений....29
8.2. Операции над типом error....29
8.3. Операции над типом enum....29
8.4. Логические выражения....29
8.4.1. Условный оператор....29
8.5. Операции над типом bit (целые числа без знака)....29
8.6. Операции над целыми числами фиксированного размера со знаком....30
8.6.1. Замечания о сдвигах....31
8.7. Операции над целыми числами произвольной разрядности....31
8.8. Операции над битовыми типами переменного размера....32
8.9. Приведение типов....32
8.9.1. Явные преобразования....32
8.9.2. Неявные преобразования....32
8.9.3. Недопустимые арифметические выражения....32
8.10. Операции над кортежами....33
8.11. Операции над списками....33
8.12. Операции над множествами....33
8.12.1. Одноэлементные множества....34
8.12.2. Универсальное множество....34
8.12.3. Маски....34
8.12.4. Диапазоны....34
8.12.5. Произведения....34
8.13. Операции над структурированными типами....34
8.14. Операции над заголовками....34
8.15. Операции над стеками заголовков....35
8.16. Операции над объединениями заголовков....36
8.17. Вызовы методов и функций....38
8.18. Вызовы конструкторов....38
9.Объявления констант и переменных....38
9.1. Константы....38
9.2. Переменные....39
9.3. Создание экземпляров....39
9.3.1. Ограничения на создание экземпляров верхнего уровня....39
10. Операторы....39
10.1. Оператор присваивания....40
10.2. Пустой оператор....40
10.3. Оператор блока....40
10.4. Оператор возврата....40
10.5. Оператор выхода....40
10.6. Оператор условия....40
10.7. Оператор вариантов....41
11. Анализ пакета....41
11.1. Состояния анализатора....41
11.2. Объявления анализаторов....41
11.3. Абстрактная машина анализа....42
11.4. Состояния анализатора....42
11.5. Операторы переходов....43
11.6. Выражения для выбора....43
11.7. Оператор проверки....44
11.8. Извлечение данных....44
11.8.1. Извлечение полей фиксированного размера....44
11.8.2. Извлечение полей переменного размера....45
11.8.3. Взгляд вперёд....46

11.8.4. Пропуск битов....46
11.9. Стеки заголовков....46
11.10. Субанализаторы....47
12. Блоки управления....47
12.1. Действия....48
12.1.1. Вызовы операций....48
12.2. Таблицы....49
12.2.1. Свойства таблиц....50
12.2.1.1. Ключи....50
12.2.1.2. Действия....50
12.2.1.3. Действие по умолчанию....51
12.2.1.4. Записи таблицы....51
12.2.1.5. Дополнительные свойства....52
12.2.2. Вызов элемента СД....53
12.2.3. Семантика выполнения блока «сопоставление-действие»....53
12.3. Абстрактная машина конвейера «сопоставление-действие»....53
12.4. Вызовы элементов управления....53
13. Параметризация....53
13.1. Непосредственный вызов типа....54
14. Сборка....54
14.1. Вставка данных в пакеты....55
15. Описание архитектуры....55
15.1. Пример описания архитектуры....55
15.2. Пример архитектурной программы....56
15.3. Модель фильтра пакетов....57
16. Абстрактная машина P4 - оценка....57
16.1. Известные при компиляции значения....57
16.2. Оценка во время компиляции....57
16.3. Имена для уровня управления....58
16.3.1. Создание имён для управления....59
16.3.1.1. Таблицы....59
16.3.1.2. Ключи....59
16.3.1.3. Действия....59
16.3.1.4. Экземпляры....59
16.3.2. Аннотации для имён управляемых элементов....60
16.3.3. Рекомендации....60
16.4. Динамическая оценка....61
16.4.1. Модель одновременной обработки....61
17. Аннотации....61
17.1. Предопределённые аннотации....61
17.1.1. Аннотации списка действий таблицы....62
17.1.2. Аннотации API уровня управления....62
17.1.2.1. Ограничения....62
17.1.3. Аннотации управления параллельной работой....62
17.2. Специфические для платформы аннотации....62
Приложение A. Зарезервированные слова P4....62
Приложение B. Библиотека ядра P4....63
Приложение C. Контрольные суммы....63
Приложение D. Нерешенные вопросы....64
D.1. Переносимая архитектура коммутатора....64
D.2. Обобщение оператора switch....64
D.3. Неопределённое поведение....64
D.4. Структурированные итерации....64
Приложение E. Грамматика P4....65

1. Назначение языка

Эта спецификация определяет структуру и интерпретацию программ на языке $P4_{16}$. В документе определён синтаксис, правила семантики и требования по совместимости для реализаций языка.

Документ не определяет:

- механизмов компиляции, загрузки и исполнения программ P4 в системах обработки пакетов;
- механизмов получения данных одной системой обработки пакетов и их доставки другой системе;
- механизмов, с помощью которых уровень управления (control plane) поддерживает таблицы «сопоставление-действие» и другие связанные с состоянием объекты, определяемые программами P4;
- минимальных требований к системам обработки пакетов, способным обеспечить реализацию.

2. Термины, определения и символы

Ниже приведены определения используемых в документе терминов.

Architecture - архитектура
Набор программируемых с помощью P4 компонент и интерфейсов уровня данных (data plane) между ними.

Control plane – уровень (плоскость) управления
Класс алгоритмов, а также соответствующих входных и выходных данных, которые связаны с подготовкой и настройкой конфигурации уровня данных.

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

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

Packet - пакет
Пакет представляет собой форматированный блок данных, передаваемый через сети с коммутацией пакетов.

Packet header – заголовок пакета
Форматированные данные в начале пакета. Конкретный пакет может содержать последовательность заголовков, представляющих различные сетевые протоколы.

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

Packet-processing system – система обработки пакетов
Система обработки данных, предназначенная для работы с сетевыми пакетами. В общем случае в системах обработки пакетов реализуются алгоритмы уровней управления и данных.

Target – целевая платформа, платформа
Система обработки пакетов, способная выполнять программы P4.

Все явно определённые в этом документе термины не следует трактовать как неявные ссылки на похожие термины, определённые в других местах. И наоборот, не определённые явно в данном документе термины следует трактовать в соответствии с общепризнанными документами, где они были определены (например, IETF RFC).

3. Обзор

Язык P4 служит для представления обработки пакетов уровнем данных в программируемых элементах сети типа аппаратных и программных коммутаторов, сетевых интерфейсных плат, маршрутизаторов или специализированных сетевых платформ. Имя P4 происходит от оригинальной публикации, в которой язык был предложен - «Programming Protocol-independent Packet Processors» (https://arxiv.org/pdf/1312.1719.pdf). Хотя P4 исходно был предназначен для программируемых коммутаторов, область его применения шире и охватывает различные устройства. Далее в документе для всех таких устройств используется общий термин «платформа» (target). Многие платформы реализуют уровни данных и управления. P4 предназначен только для работы с уровнем данных таких платформ. Программы P4 также частично определяют интерфейс, через который взаимодействуют уровни данных и управления, но P4 не может служить для описания функциональности уровня управления. Далее в документе при обсуждении использования P4 для «программирования платформы» всегда имеется в виду «программирование уровня данных платформы».

В качестве примера платформы на рисунке 1 показано различие между традиционным коммутатором с фиксированной функциональностью и программируемым на базе P4 коммутатором. В традиционном коммутаторе производитель полностью определяет функциональность уровня данных. Уровень управления контролирует уровень данных путём задания таблиц (например, таблица маршрутизации), настройки специализированных объектов (например, измерителей) и обработки управляющих пакетов (например, пакеты протоколов маршрутизации) или асинхронных событий типа изменения состояния каналов или уведомлений о получении новой информации (learning).

Программируемый на P4 коммутатор имеет два существенных отличия, показанных ниже.

- Функциональность уровня данных не фиксирована заранее, а определяется программой P4. Уровень данных настраивается во время инициализации функциональности, описанной программой P4 (длинная красная стрелка на рисунке 1), и не знает заранее о существующих сетевых протоколах.
- Уровень управления взаимодействует с уровнем данных по таким же каналам, что и в традиционном коммутаторе, но набор таблиц и других объектов уровня данных не фиксирован, поскольку он определяется программой P4. Компилятор P4 генерирует интерфейс API, используемый уровнем управления для взаимодействия с уровнем данных.

Следовательно, P4 можно считать независимым от протоколов, но он позволяет программистам выразить множество протоколов и других аспектов поведения уровня данных.

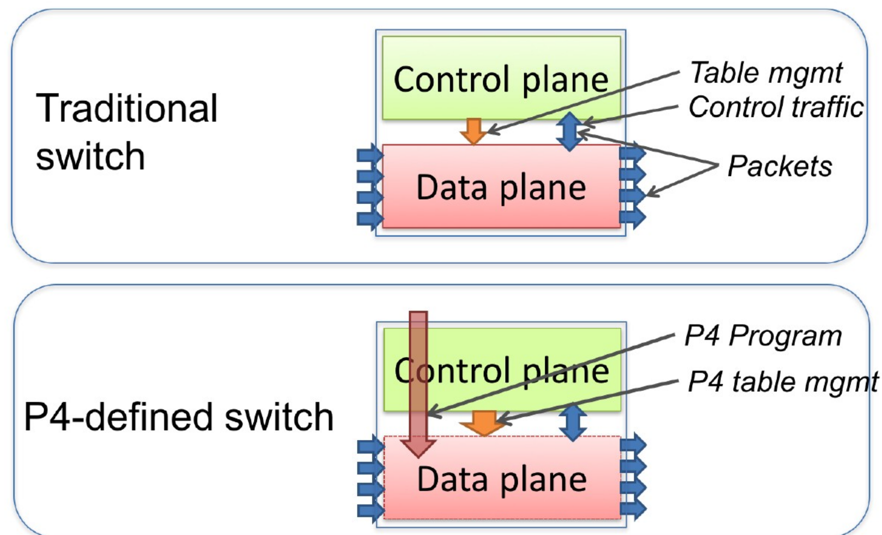

Рисунок 1. Сравнение традиционных и программируемых коммутаторов.

Ниже перечислены основные абстракции, обеспечиваемые языком программирования P4.

- Типы заголовков описывают формат (набор полей и их размеры) каждого заголовка в пакете.
- Анализаторы (parser) описывают разрешённые последовательности заголовков в принятых пакетах, способы идентификации последовательностей заголовков, а также заголовки и поля, извлекаемые из пакетов.
- Таблицы связывают заданные пользователем ключи с операциями. Таблицы P4 обобщают таблицы традиционных коммутаторов и могут служить для реализации таблиц поиска потоков, маршрутизации, списков управления доступом и других определяемых пользователем таблиц, включая комплексные решения на основе множества переменных.
- Действия являются фрагментами кода, которые описывают манипуляции с полями заголовков и метаданными. Действия могут включать данные, получаемые в процессе работы от уровня управления.
- Блоки «сопоставление-действие» выполняют приведённую ниже последовательность операций:
    - создание ключей поиска из полей пакета и рассчитанных метаданных;
    - поиск в таблицах с использованием созданных ключей и выбор действия (включая связанные данные);
    - выполнение выбранного действия.
- Поток управления выражает императивную программу, которая описывает обработку пакетов, включая зависящую от данных последовательность обращений к блокам «сопоставление-действие». В потоке управления может также выполняться сборка пакетов.
- Внешние объекты являются зависящими от архитектуры конструкциями, которыми могут манипулировать программы P4 с помощью чётко определённых API. Внутреннее поведение таких объектов задано жёстко (например, блоками контрольных сумм) и поэтому они не могут программироваться с помощью P4.
- Определяемые пользователем метаданные - структуры данных, связанные с каждым пакетом.
- Внутренние метаданные - связанные с каждым пакетом данные, предоставляемые архитектурой (например, входной порт, через который был принят пакет).

На рисунке 2 показан типичный поток операций при программировании платформы с использованием языка P4.

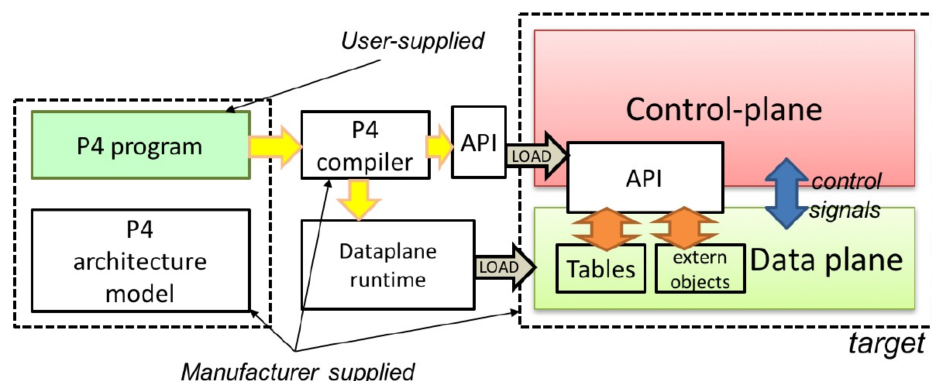

Рисунок 2. Программирование движка с помощью P4.

Производители платформ обеспечивают аппаратную и программную реализацию основы, архитектурное определение и компилятор P4 для платформы. Программисты P4 пишут программы для конкретной архитектуры, определяющие набор программируемых на P4 компонент платформы, а также внешних интерфейсов уровня данных.

Компиляция пакета программ P4 создаёт два объекта:

- конфигурацию уровня данных, которая реализует логику пересылки, описанную в программе;

- API для управления состояниями объектов уровня данных со стороны уровня управления.

Язык P4 разработан для реализации на множестве разных платформ, включая программируемые сетевые интерфейсные платы, FPGA, программные коммутаторы, контроллеры ASIC. Поэтому язык ограничивается использованием конструкций, которые можно эффективно реализовать 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 байта принятого и проанализированного пакета. Хотя анализаторы могут включать циклы, обеспечивающие извлечение данных из одного заголовка в каждом цикле, сам пакет задаёт границы для полного анализа. Иными словами, при таких допущениях расчётная сложность программы P4 линейно растёт с ростом общего размера заголовков и никогда не зависит от размера состояния, созданного в процессе обработки данных (например, число потоков или общее число обработанных пакетов). Эти гарантии необходимы (но не достаточны) для обеспечения быстрой обработки пакетов на широком спектре платформ.

Соответствие платформы требованиям P4 определяется следующим образом - если конкретная платформа T поддерживает лишь подмножество языка P4 (пусть это будет, P4T), программы, написанные в рамках P4T при выполнении на этой платформе должны обеспечивать в точности такое же поведение, какое описано в данном документе. Отметим, что соответствующие P4 платформы могут обеспечивать произвольные расширения языка P4 и внешних элементов.

3.1. Преимущества P4

По сравнению с современными системами обработки пакетов (например, основанными на микрокоде для специализированного оборудования) P4 обеспечивает значительные преимущества, перечисленные ниже.

- Гибкость - P4 создаёт множество правил пересылки пакетов в виде программ в отличие от традиционных коммутаторов, предоставляющим пользователям машины пересылки с фиксированным набором функций.
- Выразительность - P4 может выражать сложные аппаратно-независимые алгоритмы обработки пакетов, использующие только операции общего назначения и поиск в таблицах. Такие программы переносимы на разные аппаратные платформы, реализующие одну и ту же архитектуру (при наличии достаточных ресурсов).
- Отображение ресурсов и управление ими - программы P4 абстрактно описывают ресурсы хранения (например, адреса отправителей IPv4), компиляторы отображают описанные пользователем поля на доступные аппаратные ресурсы и управляют ими на нижнем уровне (например, распределение и планирование).
- Программные решения - программы P4 обеспечивают важные преимущества - проверку типов, сокрытие информации, многократное использование кода.
- Библиотеки компонент, поставляемые производителями, могут использоваться для встраивания аппаратно-зависимых функций в переносимые конструкции верхнего уровня P4.
- Независимое развитие программных и аппаратных компонент - производители платформ могут использовать абстрактную архитектуру для дальнейшего разделения деталей архитектуры на нижних уровнях и обработки на верхних.
- Отладка - производители могут предоставить программные модели архитектуры, помогающие при разработке и отлаживании программ P4.

3.2. Развитие языка P4 - сравнение с предыдущими версиями (P4 v1.0/v1.1)

По сравнению с более ранней версией $P4_{14}$ язык $P4_{16}$ вносит множество значимых и не совместимых с прежними версиями изменений в синтаксис и семантику. Переход от предыдущей версии ($P4_{14}$) к текущей ($P4_{16}$) показан на рисунке 3. В частности, из языка было изъято и перемещено в библиотеки множество функций, включая счётчики, блоки контрольных сумм, измерители и т. п.

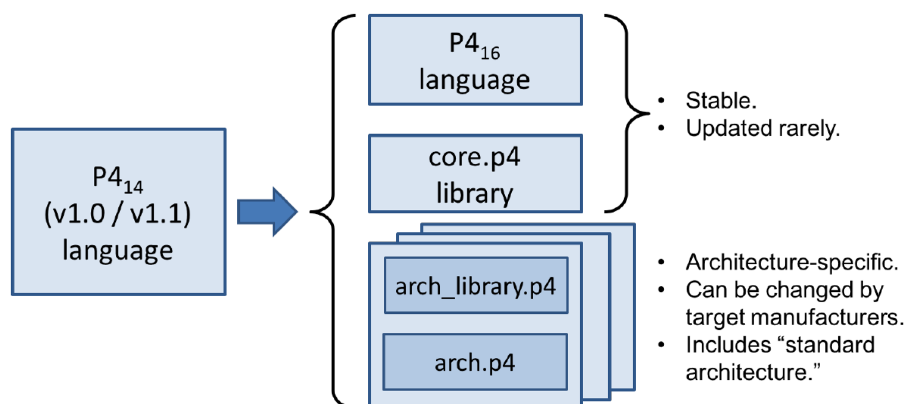

Рисунок 3. Эволюция языка от $P4_{14}$ (версии 1.0 и 1.1) до $P4_{16}$.

Язык был преобразован из сложного (более 70 ключевых слов) в сравнительно простой (менее 40 ключевых слов, описанных в Приложении A), сопровождающийся библиотекой базовых конструкции, которые нужны в большинстве программ P4.

В версии 1.1 языка P4 была введена конструкция extern, которая может служить для описания элементов библиотеки. Многие конструкции, определённые в спецификации v1, были преобразованы в библиотечные элементы (с том числе исключённые из языка конструкции типа счётчиков и измерителей). Некоторые из таких внешних (extern) объектов могут быть стандартизованы и они остаются в области действия будущего документа, описывающего стандартную библиотеку элементов P4. В этом документе представлено несколько примеров конструкций extern. В $P4_{16}$ также

добавлены или переопределены некоторые языковые конструкции v1.1 для описания программируемых частей архитектуры. Эти конструкции включают parser (анализатор), state (состояние), control (элемент управления) и package (программный пакет).

Одной из важных целей при подготовке спецификации $P4_{16}$ было определение стабильного языка. Иными словами, мы стремились обеспечить, чтобы все программы $P4_{16}$ оставались синтаксически корректными и вели себя неизменно в будущих версиях языка. Кроме того, при появлении в будущих версиях языка несовместимых со старыми элементов или конструкций мы будем искать простой путь перевода программ $P4_{16}$ на новую версию.

4. Модель архитектуры

Архитектура P4 задаёт программируемые на языке P4 блоки (например, анализатор, входной и выходной поток управления и т. п.) и их интерфейсы уровня данных.

Архитектуру P4 можно рассматривать как соглашение между программой и платформой. Поэтому каждый производитель должен предоставлять компилятор P4 с определением архитектуры для своей платформы (предполагается, что компиляторы P4 смогут использовать общий пользовательский интерфейс для всех вариантов архитектуры). Определение архитектуры не обязано раскрывать все программируемые элементы уровня данных, производитель может даже предоставлять на выбор множество определений одного аппаратного элемента с разными возможностями (например, с поддержкой групповой адресации и без неё).

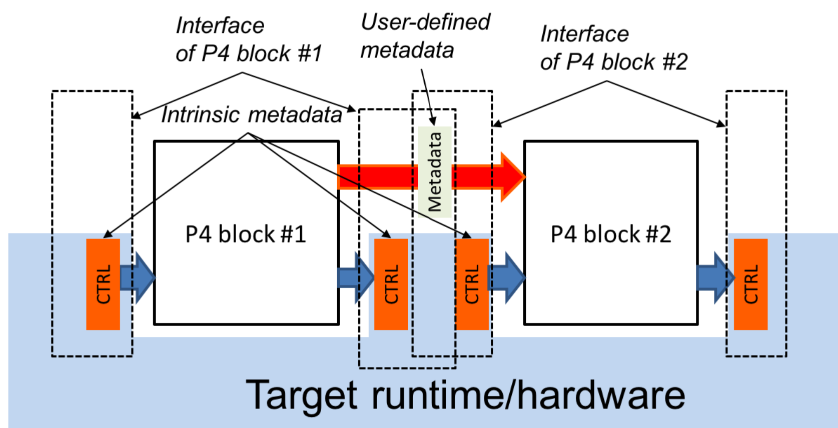

Рисунок 4. Программные интерфейсы P4.

На рисунке 4 показаны интерфейсы уровня данных между программируемыми на языке P4 блоками. Платформа имеет два программируемых блока (#1 и #2), каждый из которых задаётся отдельным фрагментом кода P4. Платформа взаимодействует с программами P4 через набор управляющих регистров или сигналов. Входные элементы управления предоставляют информацию программам P4 (например, порт, принявший пакет), а выходные могут задаваться программами P4 для влияния на поведение платформы (например, выбор выходного порта для пакета). Регистры управления и сигналы представляются в P4 как внутренние метаданные. Программы P4 могут также хранить и обрабатывать относящиеся к каждому пакету метаданные, определяемые пользователем.

Поведение программы P4 можно полностью описать в терминах преобразований, отображающих векторы битов на другие битовые векторы. Для реальной обработки пакета модель архитектуры интерпретирует биты, которые программа P4 записывает во внутренние метаданные. Например для пересылки пакета в определённый выходной порт программе P4 может потребоваться записать индекс выходного порта в определённый регистр управления. Аналогично, для отбрасывания пакета программе P4 может потребоваться установить флаг drop в другом регистре управления. Отметим, что детали интерпретации внутренних метаданных зависят от архитектуры.

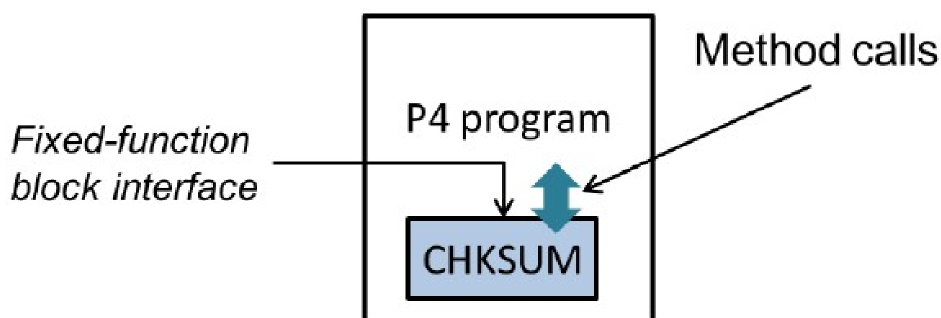

Рисунок 5. Вызов программой P4 службы объекта с фиксированными функциями.

Программы P4 могут обращаться к службам, реализованным внешними объектами, и обеспечиваемым архитектурой функциям. На рисунке 5 программа P4 вызывает встроенную услугу расчёта контрольной суммы целевой платформы. Реализация модуля контрольных сумм не задаётся в P4, но интерфейс с этим модулем задан. В общем случае интерфейс для внешнего объекта описывает каждую поддерживаемую им операцию, а также параметры и типы возвращаемых значений.

В общем случае программы P4 не предполагаются переносимыми между разными платформами. Например, программа P4, которая передаёт широковещательные пакеты путём записи в регистр управления, не будет корректно работать на платформах без такого регистра. Однако программы P4, написанные для данной архитектуры, должны быть переносимы между разными платформами, которые корректно реализуют соответствующую модель и имеют достаточно ресурсов.

4.1. Стандартные модели архитектуры

Предполагается, что сообщество P4 разработает небольшой набор стандартных архитектурных моделей для конкретных вертикалей. Широкое распространение таких моделей будет способствовать переносимости программ P4 между различными платформами. Определение стандартных вариантов архитектуры выходит за рамки документа.

4.2. Интерфейсы уровня данных

Для описания функциональных блоков, которые могут быть запрограммированы в P4, архитектура включает объявление типа, которое задаёт интерфейсы между этим блоком и другими компонентами архитектуры. Например, архитектура может включать определение вида

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

Этот тип объявления описывает блок MatchActionPipe, который можно запрограммировать с помощью зависящей от данных последовательности вызовов блоков «сопоставление-действие» (далее СД) и других обязательных конструкций, указанных ключевым словом `control`. Интерфейс между блоком MatchActionPipe и другими компонентами архитектуры можно определить из объявления блока.

- Первым параметром служит 4-битовое значение inputPort. Направление in говорит, что параметр является входным и не может быть изменён.
- Вторым параметром является объект типа H с именем parsedHeaders, где H - переменная типа, представляющая заголовки (будет определена ниже). Направление inout говорит, что параметр является входным и выходным.
- Третьим параметром является 4-битовое значение outputPort. Направление out говорит, что параметр является выходным и его значение, заданное изначально, может быть изменено.

4.3. Внешние объекты и функции

Программы P4 могут взаимодействовать с объектами и функциями, обеспечиваемыми архитектурой. Такие объекты описываются с помощью конструкции extern, которая задаёт интерфейсы, раскрываемые объектом уровню данных.

Объект extern описывает набор методов, реализуемых этим объектом, но не сами реализации (подобно абстрактным классам в объектно-ориетированных языках). Например, приведённая ниже конструкция может служить для описания операций блока инкрементного расчёта контрольных сумм.

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

5. Пример очень простого коммутатора

В качестве примера, показывающего возможности архитектуры, рассмотрим реализацию очень простого коммутатора с помощью P4. Сначала будет описана архитектура коммутатора, а затем приведён полный код программы P4, которая задаёт поведение уровня данных этого коммутатора. Этот пример иллюстрирует многие важные функции языка P4.

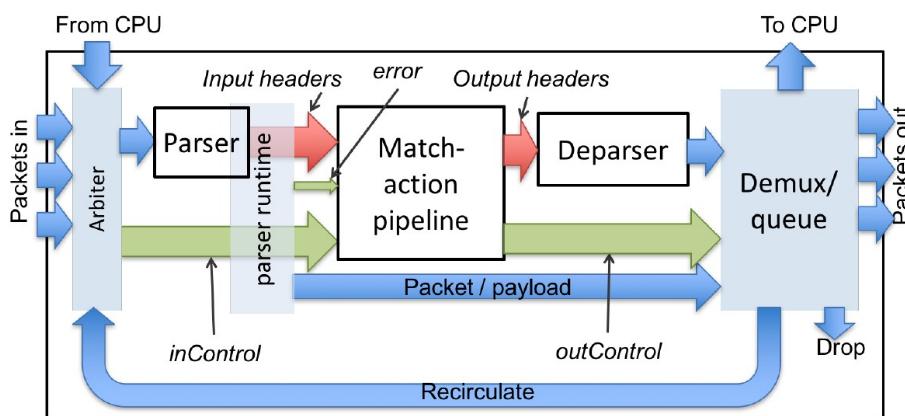

Рисунок 6. Архитектура очень простого коммутатора.

Назовём эту архитектуру VSS[1] (смс рисунок 6). В VSS нет ничего особенного - это всего лишь дидактический пример, иллюстрирующий описание и программирование коммутатора с помощью языка P4. VSS имеет множество фиксированных функциональных блоков (на рисунке показаны светло-синим цветом), поведение которых описано в параграфе 5.2. Белые блоки на рисунке программируются с помощью P4.

VSS принимает пакеты через один из 8 входных портов Ethernet, канал рециркуляции или порт, напрямую подключенный к процессору CPU. VSS имеет один синтаксический анализатор, результаты которого передаются в единственный конвейер СД, а за ним следует единственный синтезатор (deparser). После выхода из синтезатора пакет передаётся в один из 8 выходных портов Ethernet или 3 портов специального назначения:

- порт CPU для передачи на уровень управления;

[1]Very Simple Switch - очень простой коммутатор.

- порт Drop для отбрасывания пакета;
- порт рециркуляции (Recirculate) для последующего возврата в коммутатор через специальный входной порт.

Белые блоки на рисунке являются программируемыми и пользователь должен предоставить соответствующую программу P4 для задания поведения такого блока. Красные стрелки показывают поток определённых пользователем данных. Синие блоки являются компонентами с фиксированной функциональностью. Зелёные стрелки являются интерфейсами уровня данных, используемыми для обмена информацией между фиксированными функциональными блоками и программой P4 в форме внутренних метаданных.

5.1. Архитектура очень простого коммутатора

Приведённая ниже программа P4 содержит объявление VSS в P4, представленное производителем VSS. Объявление включает несколько объявлений типов и трёх программируемых блоков (в коде используется цветовое выделение). Программируемые блоки описываются их типами, реализация этих блоков обеспечивается программистами.

```
// Файл very_simple_switch_model.p4
// Объявление очень простого коммутатора VSS P4

// Библиотека ядра, требуемая для определений packet_in и packet_out
# include <core.p4>

/* Объявления констант и структур */
/* Порты указываются 4-битовыми значениями */
typedef bit<4> PortId;

/* Только 8 портов являются физическими */
const PortId REAL_PORT_COUNT = 4w8; // 4w8 представляет 4-битовое число 8

/* Метаданные, сопровождающие входной пакет */
struct InControl {
   PortId inputPort;
}

/* Входные порты специального назначения */
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;

/* Метаданные, которые нужно рассчитать для выходных пакетов */
struct OutControl {
   PortId outputPort;
}

/* Выходные порты специального назначения для исходящих пакетов */
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;

/* Прототипы для всех программируемых блоков */
/**
* Программируемый синтаксический анализатор.
* @param <H> - тип заголовков, указываемый пользователем;
* @param b - входной пакет;
* @param parsedHeaders заголовки, созданные анализатором.
*/
parser Parser<H>(packet_in b,out H parsedHeaders);

/**
* Конвейер «сопоставление-действие» (СД)
* @param <H> - тип входных и выходных заголовков;
* @param headers - заголовки, получаемые от анализатора и передаваемые синтезатору;
* @param parseError - ошибка, которая могла возникнуть в процессе анализа;
* @param inCtrl - информация от архитектуры, сопровождающая входной пакет;
* @param outCtrl - информация от 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

/**
* Синтезатор (deparser) VSS.
* @param <H> - тип заголовков, указываемый пользователем;
* @param b - выходной пакет;
* @param outputHeaders - заголовки для выходного пакета.
*/
control Deparser<H>(inout H outputHeaders, packet_out b);

/**
* Объявление верхнего уровня для программы (экземпляр должен быть создан пользователем).
* Аргументы программы указывают блоки, экземпляры которых нужно создать пользователю.
* @param <H> - определённый пользователем тип обрабатываемых заголовков.
*/
package VSS<H>(Parser<H> p,
                 Pipe<H> map,
                 Deparser<H> d);
```

```
// Определяемые архитектурой объекты, экземпляры которых могут быть созданы.

// Блок контрольных сумм.
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 имеющейся контрольной суммы
   bit<16> get();                // получение контрольной суммы для данных, добавленных
                                 // с момента последней очистки
}
```

Далее описаны некоторые из элементов.

- Включаемый файл core.p4 более подробно рассмотрен в Приложении B. Он определяет некоторые стандартные типы данных и коды ошибок.
- bit<4> - битовая строка из 4 битов.
- Синтаксис 4w0xF показывает значение 15, представленное 4 битами. Другим вариантом записи является 4w15. Во многих случаях размер можно опустить, просто указав 15.
- error - это встроенный тип P4 для передачи кодов ошибок.
- Далее следует объявление анализатора

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

  Это объявление описывает интерфейс анализатора, но не его реализацию, которая будет задана программистом. Анализатор считывает свои входные данные из packet_in (предопределённый внешний объект P4, представляющий входящий пакет и описанный в библиотеке core.p4). Анализатор записывает свои результаты (ключевое слово out) в аргумент parsedHeaders. Типом этого аргумента является H, который ещё не известен и будет предоставлен программистом.
- Объявление

  ```
  control Pipe<H>(inout H headers,
                  in error parseError,       // ошибка анализатора
                  in InControl inCtrl,       // входной порт
                  out OutControl outCtrl);   // выходной порт
  ```

  описывает интерфейс с конвейером СД, названным Pipe.

  Конвейер принимает 3 входных параметра - заголовки headers, код ошибок при анализе parseError и данные управления inCtrl. На рисунке 6 показаны разные источники этих частей информации. Конвейер записывает свои результаты в outCtrl и должен обновить заголовки, которые будут переданы синтезатору (deparser).
- Программа верхнего уровня называется VSS. Для программирования коммутатора VSS пользователь будет создавать экземпляр программы этого типа (см. следующий параграф). Объявление программы верхнего уровня также зависит от типа переменной H

  ```
  package VSS<H>
  ```

  Тип переменной ещё не известен и должен быть представлен пользователем позднее. В данном случае H является типом из набора заголовков, которые пользовательская программа будет обрабатывать. Анализатор будет разбирать эти заголовки, а конвейер СД - обновлять входные заголовки для создания выходных.
- Объявление программы VSS включает три комплексных параметра типов Parser, Pipe и Deparser, которые точно соответствуют описанным выше объявлениям. Для программирования платформы нужно будет представить значения этих трёх параметров.
- В этой программе структуры inCtrl и outCtrl представляют регистры управления. Содержимое структуры headers сохраняется в регистрах общего назначения.
- Объявление extern Checksum16 описывает внешний объект, который может быть вызван для расчёта контрольных сумм.

5.2. Описание архитектуры VSS

Чтобы полностью понять поведение коммутатора VSS и написать для него осмысленные программы P4, а также реализовать уровень управления, потребуется также полное описание поведения фиксированных функциональных блоков. В этом параграфе рассматривается простой пример, показывающий все детали, которые нужно учитывать при описании архитектуры. Язык P4 не предназначен для описания всех таких функциональных блоков, он может описать лишь интерфейсы между программируемыми блоками и архитектурой. Для нашей программы этот интерфейс задаётся объявлениями Parser, Pipe и Deparser. На практике предполагается полное описание архитектуры в форме исполняемой программы и/или рисунков и текста, а данный документ даёт лишь неформальное текстовое описание.

5.2.1. Блок арбитража

Входной блок арбитража выполняет перечисленные ниже функции.

- Приём пакетов от одного из физических портов Ethernet, уровня управления или входного порта рециркуляции.
- Для пакетов из портов Ethernet блок рассчитывает и проверяет контрольную сумму трейлера и при несоответствии отбрасывает пакет. Если контрольная сумма верна, она удаляется из данных пакета.
- При получении пакетов включается алгоритм арбитража, если доступно множество пакетов одновременно.
- Если блок арбитража занят обработкой предыдущего пакета и в очереди нет свободного места, входной порт может отбрасывать прибывающие пакеты без какой-либо индикации таких событий.

- После приёма пакета блок арбитража устанавливает значение inCtrl.inputPort, которое служит входными данными для анализатора, указывая порт, через который был получен пакет. Физические порты Ethernet имеют номера от 0 до 7, входной порт рециркуляции имеет номер 13, а порт процессора - 14.

5.2.2. Блок анализатора

Блок parser runtime работает вместе с блоком синтаксического анализатора и предоставляет код ошибки конвейеру СД на основе действий анализатора, а также информацию о данных (payload) пакета (например, размер оставшейся части данных) блоку демультиплексирования (demux). По завершении обработки пакета анализатором в процесс вовлекается конвейер СД, принимая на вход метаданные (заголовки пакета и заданные пользователем метаданные).

5.2.3. Блок демультиплексирования

Основной функциональностью блока demux является получение заголовков для выходных пакетов от синтезатора и данных пакета от анализатора для сборки из них нового пакета и отправки результата в нужный выходной порт, который задаётся значением outCtrl.ouputPort, устанавливаемым конвейером СД.

- Передача пакета в порт drop приводит к исчезновению пакета.
- Передача пакета в выходной порт Ethernet с номером от 0 до 7 вызывает отправку данных через соответствующий выходной интерфейс. Пакет может быть помещён в очередь, если выходной интерфейс уже занят передачей другого пакета. При отправке пакета физический интерфейс рассчитывает контрольную сумму трейлера Ethernet и добавляет её в конце пакета.
- Передача пакета в выходной порт CPU приводит к пересылке этого пакета уровню управления. В этом случае процессору CPU передаётся исходный пакет, а не пакет, полученный от синтезатора (он отбрасывается[1]).
- Передача пакета в выходной порт рециркуляции приводит в конечном итоге к появлению этого пакета на входном порту рециркуляции. Рециркуляция полезна в тех случаях, когда обработку пакета невозможно выполнить за один проход.
- Если outputPort имеет недопустимое значение (например, 9), пакет отбрасывается.
- Если блок demux занят обработкой предыдущего пакета и нет возможности поместить приходящий от синтезатора пакет в очередь, этот пакет отбрасывается независимо от указанного выходного порта.

Следует обратить внимание на то, что некоторые действия блока демультиплексирования могут быть неожиданными (они будут выделяться жирным шрифтом). Здесь не рассматриваются некоторые важные вопросы поведения, связанные с очередями, арбитражем и синхронизацией, которые тоже влияют на обработку пакета.

Зелёная стрелка на рисунке 5 от блока parser runtime к блоку demux представляет дополнительный поток информации от анализатора к демультиплексору - обрабатываемый пакет, а также смещение в пакете, на котором был закончен анализ (т. е. начало данных пакета).

5.2.4. Доступные внешние блоки

Архитектура VSS поддерживает внешний блок инкрементного расчёта контрольных сумм Checksum16. Блок контрольных сумм включает конструктор и четыре метода.

- `clear()` - подготовка блока к новому расчёту.
- `update<T>(in T data)` - добавление данных к учитываемым в контрольной сумме. Данные должны быть строкой битов, типом заголовка или конструкцией из этих двух типов. Поля в заголовке/структуре объединяются (конкатенация) в порядке их указания в определении типа.
- `get()` - возвращает 16-битовую контрольную сумму с дополнением до 1. При вызове этой функции контрольная сумма должна учитывать целое число байтов данных.
- `remove<T>(in T data)` - в предположении, что некие данные были учтены в контрольной сумме удаляет их из расчёта.

[1]В простейшем случае коммутатора VSS это верно, но в реальности пакет может сохраняться и обрабатываться дальше с передачей в обычный выходной порт или порт рециркуляции. *Прим. перев.*

5.3. Полная программа VSS

Здесь представлена полная программа P4, которая реализует базовую пересылку пакетов IPv4 на основе архитектуры

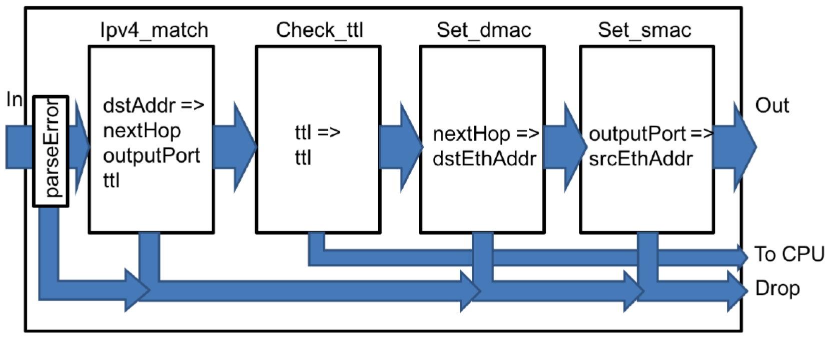

Рисунок 7. Конвейер таблиц СД программы VSS P4.

VSS. Эта программа не использует всех функций, обеспечиваемых архитектурой (например, рециркуляция), но применяет директивы препроцессора #include (см. параграф 6.2).

Анализатор пытается распознать заголовок Ethernet, за которым следует заголовок IPv4. При отсутствии любого из этих заголовков возникает ошибка. В остальных случаях информация из заголовков передаётся в структуру Parsed_packet. Конвейер СД, показанный на рисунке 7, состоит из 4 блоков, обозначенных именами таблиц P4.

- При возникновении ошибки анализатора пакет отбрасывается путём установки outputPort = DROP_PORT.
- Первая таблица использует адрес получателя IPv4 для определения outputPort и адреса IPv4 следующего маршрутизатора (next hop). Если поиск не дал результата, пакет отбрасывается. Таблица также декрементирует значение IPv4 ttl.
- Вторая таблица проверяет значение ttl и при ttl=0 пакет пересылается на уровень управления через порт CPU.
- Третья таблица использует адрес IPv4 следующего маршрутизатора (его даёт первая таблица) для определения адреса Ethernet следующего маршрутизатора.
- Последняя таблица использует outputPort для идентификации адреса отправителя Ethernet в текущем коммутаторе, который будет указан в выходном пакете.

Синтезатор создаёт исходящий пакет, собирая заголовки Ethernet и IPv4, рассчитанные конвейером.

```
// Включение библиотеки ядра P4
# include <core.p4>

// Включение файла с объявлениями архитектуры VSS.
# include "very_simple_switch_model.p4"

// Эта программа обрабатывает пакеты, включающие заголовки Ethernet и IPv4,
// и пересылает 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 разобранных заголовков
struct Parsed_packet {
   Ethernet_h  ethernet;
   IPv4_h         ip;
}
// Раздел анализатора

// Определённые 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 остальные 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 того, что контрольная сумма пакета равна 0
   verify(ck.get() == 16w0, error.IPv4ChecksumError);
   transition accept;
}

// Раздел конвейера СД
control TopPipe(inout Parsed_packet headers,
                  in error parseError, // Ошибка анализатора
                  in InControl inCtrl, // Входной порт
                  out OutControl outCtrl) {
      IPv4Address nextHop;              // Локальная переменная

      /**
      * Указывает, что пакет отбрасывается путём установки
      * выходного порта DROP_PORT
      */
      action Drop_action() {
         outCtrl.outputPort = DROP_PORT;
      }

      /**
      * Устанавливает next hop и выходной порт.
      * Декрементирует поле ipv4 ttl.
      * @param ivp4_dest - адрес ipv4 следующего маршрутизатора;
      * @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;
      }

      /**
      * Определяет адрес IPv4 следующего маршрутизатора и выходной порт
      * на основе адреса получателя IPv4 в текущем пакет.
      * Декрементирует IPv4 TTL в пакете.
      * @param nextHop - адрес IPv4 следующего маршрутизатора.
      */
      table ipv4_match {
         key = { headers.ip.dstAddr: lpm; } // Максимальное совпадение префиксов
         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 CPU устаревших пакетов. */
      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 в пакет
      * @param dmac - MAC-адрес получателя.
```

```
      */
      action Set_dmac(EthernetAddress dmac) {
         headers.ethernet.dstAddr = dmac;
      }

      /**
      * Установка Ethernet адреса получателя пакета
      * на основе IP-адреса следующего маршрутизатора.
      * @param nextHop - адрес IPv4 следующего маршрутизатора.
      */
      table dmac {
         key = { nextHop: exact; }
         actions = {
            Drop_action;
            Set_dmac;
         }
         size = 1024;
         default_action = Drop_action;
      }

      /**
      * Установка MAC-адреса отправителя.
      * @param smac - используемый 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();    // Прямой вызов операции отбрасывания
         return;
      }
      ipv4_match.apply(); // Результат сопоставления будет давать 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 синтезатора
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);
   }
}

// Экземпляр программы VSS верхнего уровня
VSS(TopParser(),
   TopPipe(),
   TopDeparser()) main;
```

6. Определение языка P4

Язык P4 можно рассматривать как несколько отдельных компонент, перечисленных ниже.

- Ядро, включающее типы, переменные, область действия, объявления, операторы, выражения и т. п.
- Субязык для анализаторов на основе конечного автомата состояний (раздел 11).
- Субязык для расчётов, использующих блоки СД на основе традиционного императивного потока управления (раздел 12).
- Субязык для описания архитектуры (раздел 15).

6.1. Синтаксис и семантика

6.1.1. Грамматика

Полная грамматика $P4_{16}$ представлена в Приложении E с использованием языка описания грамматики Yacc/Bison. Приведённый здесь текст основан на той же грамматике. При описании используются указанные ниже стандартные соглашения:

- символы ВЕРХНЕГО регистра означают терминалы[1] грамматики;
- выдержки из грамматики используют нотацию BNF, как показано ниже.

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

Псевдокод (используемый в основном для описания семантики различных конструкций P4) набран шрифтом фиксированного размера, как в приведённом ниже примере.

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

6.1.2. Семантика и абстрактные машины P4

Мы описываем семантику P4 в терминах абстрактных машин, выполняющих традиционный императивный код. Имеется абстрактная машина для каждого субязыка P4 (анализатор, элемент управления). Абстрактные машины описываются с помощью псевдокода.

Компиляторы P4 могут свободно реорганизовать генерируемый ими код при условии, что видимое извне поведение программ P4 соответствует описанному в этой спецификации. Видимое извне поведение включает:

- поведение ввода и вывода всех блоков P4;
- состояние, поддерживаемое внешними блоками.

6.2. Предварительная обработка

Для обеспечения сборки программ из множества исходных файлов компиляторам P4 следует поддерживать функциональность перечисленных ниже директив препроцессора C:

```
#define для определения макросов (без аргументов)
#endif
#if #else #endif #ifdef #ifndef #elif
#include
```

Препроцессору также следует удалять последовательности символов \ и перевода строки (коды ASCII 92, 10) для объединения многочисленных строк, созданных при форматировании текста программ для удобства восприятия.

Могут поддерживаться также дополнительные возможности препроцессоров C (например, макросы с аргументами), но это не гарантируется. Подобно принятой в C практике, директива #include может задавать имена файлов в двойных кавычках или в угловых скобках <>.

```
# include <системный файл>
# include "пользовательский файл"
```

Различие между этими формами заключается в порядке поиска заголовочных файлов, когда путь не задан полностью.

Компиляторам P4 следует корректно обрабатывать директивы #line, которые могут создаваться в процессе предварительной обработки. Эта функциональность позволяет собирать программы P4 из множества исходных файлов, которые могли создаваться разными программистами в разное время, включая:

- библиотеку ядра P4, определённую в этом документе;
- архитектуру, определяющую интерфейсы уровня данных и внешние блоки;
- пользовательские библиотеки компонент (например, определения заголовков стандартных протоколов);
- программы P4, задающие поведение каждого программируемого блока.

6.2.1. Библиотека ядра P4

Спецификация языка P4 определяет библиотеку ядра, включающую несколько программных конструкций общего назначения. Описание библиотеки представлено в Приложении B. Все программы P4 должны включать библиотеку ядра с помощью директивы

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

6.3. Лексические конструкции

Во всех ключевых словах P4 используются только символы ASCII. Все идентификаторы P4 должны использовать только символы ASCII. Компиляторам P4 следует корректно обрабатывать строки с 8-битовыми символами в комментариях и строковых литералах. Строчные и прописные буквы в P4 различаются. Пробельные символы, включая перевод строки, считаются разделителями. Отступы от начала строки не регламентируются, однако в P4 используются

[1]Объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение. *Прим. перев.*

похожие на C конструкции блоков и во всех примерах используются принятые в C отступы. Символы табуляции трактуются как пробелы.

Лексический анализатор (lexer) распознает следующие типы терминалов:

- IDENTIFIER (идентификатор) - начинается с буквы или символа подчёркивания и может содержать буквы, цифры и символы подчёркивания;
- TYPE (тип) - идентификатор названия типа;
- INTEGER (целое число) - целочисленные литералы;
- DONTCARE - одиночный символ подчёркивания;
- ключевые слова типа RETURN - по соглашению каждый терминал ключевого слова соответствует ключевому слову языка, набранному символами 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 символа подчёркивания. Специальный идентификатор, включающий лишь символа подчёркивания (_), указывает значение don't care (не важно), тип которого может зависеть от контекста. Некоторые ключевые слова (например, apply) могут использоваться в качестве идентификаторов, если не возникает неоднозначности с учётом контекста.

```
nonTypeName
        : IDENTIFIER
        | APPLY
        | KEY
        | ACTIONS
        | STATE
        ;
name
        : nonTypeName
        | TYPE
        ;
```

6.3.2. Комментарии

Язык P4 поддерживает несколько типов комментариев:

- однострочный комментарий, начинающийся с символов // и продолжающийся до конца строки;
- многострочный комментарии, заключенный между символами /* и */;
- вложенные многострочные комментарии не поддерживаются;
- комментарии в стиле Javadoc, начинающиеся символами /** и заканчивающиеся */

Применение комментариев Javadoc настоятельно рекомендуется для таблиц и действий, которые используются при создании интерфейса с уровнем управления.

P4 трактует комментарии как разделители, поэтому не допускается включение комментариев в маркеры (например, bi/**/t будет трактоваться как два маркера bi и t, , не bit).

6.3.3. Константы

6.3.3.1. Логические константы

Для логического типа Boolean имеется две константы - true и false.

6.3.3.2. Целочисленные константы

Целочисленные константы указываются положительными[1] целыми числами с произвольной разрядностью (arbitrary-precision). По умолчанию литералы используют десятичное представление. Для других оснований в константах приняты префиксы:

- 0x или 0X указывает шестнадцатеричные числа (основание 16);
- 0o или 0O указывает восьмеричные числа (основание 8);
- 0b или 0B указывает двоичные числа (основание 2).

Размер числовых констант в битах (разрядность) может быть указан целым числом без знака, за которым следует идентификатор наличия знака у самой константы:

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

Отметим, что 0 в начале не означает восьмеричной константы. Символ подчёркивания внутри константы считается цифрой, но игнорируется при анализе значения. Это позволяет сделать длинные константы более читаемыми за счёт группировки цифр. Символ подчёркивания нельзя использовать в спецификации размерности или в качестве первого символа целочисленного литерала. Ниже приведены примеры целочисленных констант.

```
32w0xFF        // 32-битовое целое число без знака со значением 255
32s0xFF        // 32-битовое целое число со знаком и значением 255
8w0b10101010        // 8-битовое целое число без знака со значением 0xAA
```

[1]Это означает отсутствие знака «-» перед записью константы, но не препятствует заданию отрицательных чисел. Наличие знака у константы определяется префиксом w или s, а сам знак - старшим битом константы. *Прим. перев.*

```
8w0b_1010_1010           // 8-битовое целое число без знака со значением 0xAA
8w170                    // 8-битовое целое число без знака со значением 0xAA
8s0b1010_1010            // 8-битовое целое число со знаком и значением -86
16w0377          // 16-битовое целое число без знака со значением 377 (не 255!)
16w0o377                 // 16-битовое целое число без знака со значением 255 (основание 8)
```

6.3.3.3. Строковые литералы

Строковые литералы (символьные константы) представляют собой произвольные последовательности 8-битовых символов, заключённых в двойные кавычки " (код ASCII 34). Строка начинается с символа " и продолжается до первого символа ", перед которым нет нечётного количества символов обратной дробной черты \ (код ASCII 92). P4 не проверяет корректность символьных строк (т. е. не проверяет корректность кодировки UTF-8).

Поскольку P4 не выполняет каких-либо операций со строками, строковые литералы обычно передаются компилятором P4 без изменения другим инструментам или компиляторам, включая сохранение кавычек. Эти инструменты могут определять свою трактовку escape-последовательностей (например, для задания символов Unicode или обработки не печатаемых символов ASCII).

Ниже приведены три примера строковых литералов.

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

6.4. Соглашения об именовании

P4 поддерживает множество типов. Базовые типа включают битовые строки (bit-string), целые числа (number) и ошибки (error). Имеются также встроенные типы для представления конструкций типа анализаторов (parser), конвейеров (pipeline), действий (action) и таблиц (table). Пользователи могут создавать новые типы с применением структур (structure), перечислений (enum), заголовков (header), стеков заголовков (header_stack), объединений заголовков (header_union) и т. п.

В этом документе применяются следующие соглашения:

- встроенные типы указываются символами нижнего регистра (например, int<20>);
- в пользовательских типах применяются заглавные буквы (например, IPv4Address);
- переменные типов всегда указываются заглавными буквами (например, parser P<H, IH>(…));
- в именах переменных заглавные буквы не применяются (например, ipv4header);
- имена констант указываются заглавными буквами (например, CPU_PORT);
- ошибки и перечисляемые (enum) указываются в «верблюжьем» стиле (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
        ;
```

Пустые объявления указываются одним символом точки с запятой (;). Это учитывает привычки программистов C/C++ и Java (например, в некоторых конструкциях типа struct не требуется указывать точку с запятой в конце).

6.5.1. Области действия

Некоторые конструкции P4 выступают в качестве пространства имён с локальной значимостью, которое включает:

- объявления производных типов (struct, header, header_union, enum);
- операторы блоков, которые ограничивают локальную область действия;
- блоки parser, table, action и control с локальной значимостью;
- объявления с типами переменных, которые задают новую область действия для этих переменных; например, приведённое ниже объявление extern задаёт область действия переменной типа H до конца объявления

```
extern E<H>(...) { ... } // Область действия H заканчивается здесь.
```

Порядок объявления важен. За исключением состояний анализатора все применения символа должны следовать его объявлению (это отличается от P4$_{14}$, где порядок объявления был не важен). Данное требование существенно упрощает реализацию компиляторов P4, позволяя им использовать дополнительную информацию об объявленных идентификаторах для устранения неоднозначностей.

6.5.2. Элементы с внутренними состояниями

Большинство конструкций P4 не имеет состояний (stateless) - при некой входной информации они выдают результат, определяемый исключительно этой информацией. Имеется лишь две конструкции которые могут сохранять информацию о состоянии:

- table - таблицы доступны уровню данных только для чтения, но их элементы могут изменять уровень управления;
- объекты extern - многие из таких объектов имеют состояние, которое поддерживает чтение и запись со стороны уровней данных и управления. Все конструкции языка $P4_{14}$, которые инкапсулируют состояние (например, счётчики, измерители, регистры) представляются в $P4_{16}$ с использованием объектов extern.

В P4 все элементы с внутренними состояниями (stateful) должны явно выделяться во время компиляции с помощью процесса создания экземпляра (instantiation).

Кроме того, анализаторы, блоки управления и программы могут включать создание экземпляров элементов с внутренними состояниями. Таким образом, они также считаются элементами с внутренним состоянием (даже если они не показывают наличие этого состояния) и их экземпляры должны явно создаваться до начала использования. Однако таблицы, хотя они и имеют внутренние состояния, ведут себя иначе - экземпляр таблицы создаётся при явном объявлении этой таблицы. Для тонкого контроля над созданием экземпляров таблиц программист может объявлять их внутри элементов управления.

В примере из параграфа 5.3 TopParser, TopPipe, TopDeparser, Checksum16 и Switch являются типами. Имеется два экземпляра Checksum16 (один в TopParser, другой в TopDeparser), обозначенные ck. Экземпляры TopParser, TopDeparser, TopPipe и Switch создаются в конце программы при объявлении основного объекта (main), который является экземпляром типа Switch (программа).

6.6. L-значения

L-value представляет собой выражение, которое может присутствовать в левой части операций присваивания или в качестве аргумента, соответствующего параметрам функции out или inout. Эти значения представляют ссылки на хранилища. Приведённые ниже выражения являются допустимыми l-значениями.

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

- идентификаторы базового или производного типа;
- операции доступа к полям структуры, заголовка или объединения заголовков (нотация с точками);
- ссылки на элементы стека заголовков (параграф 8.15), индекс или указание последнего (last) или следующего (next);
- результат выборки битов[m:l].

Примером допустимого l-значения может служить headers.stack[4].field. Отметим, что вызовы методов и функций не могут возвращать l-значений.

6.7. Соглашения о вызовах - copy in/copy out

P4 поддерживает множество конструкций для создания модульных программ - внешние методы extern, анализаторы, элементы управления, действия. Все эти конструкции ведут себя как процедуры языков программирования общего назначения.

- Именованные и типизованные параметры.
- Создание новой области действия для параметров и локальных переменных.
- Возможность передачи аргументов путём их привязки к параметрам.

Вызовы выполняются с использованием семантики copy-in/copy-out.

С каждым параметром может быть связана метка направления, как описано ниже.

- Параметры in предназначены только для чтения. Будет ошибкой использование параметра in в левой части оператора присваивания или его передача вызываемому в качестве аргумента не in. Параметры in создаются путём копирования значения соответствующего аргумента при вызове.
- Параметры out не инициализируются (параметры типа header или header_union устанавливаются «непригодными») и трактуются как l-значения (параграф 6.6) внутри метода или функции. Аргументы, передаваемые в качестве параметров out, должны быть l-значениями. После выполнения вызова значение параметра копируется в соответствующее место хранилища для данного l-значения.
- Параметры inout являются входными и выходными. Аргумент, передаваемый в качестве параметра inout, должен быть l-значением.
- Отсутствие направления говорит от том, что параметр является одним из перечисленных:
    - значение, известное в момент компиляции;

- параметр действия, который может быть установлен только уровнем управления;
- параметр действия, который может быть напрямую установлен другим вызываемым действием (в этом случае он ведёт себя как параметр in).

Аргументы оцениваются слева направо до вызова функции. Порядок оценки важен в тех случаях, когда представленное для аргумента выражение может иметь побочные эффекты. Для этого рассмотрим пример

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

Обратите внимание, что оценка g может изменить его аргумент a, поэтому компилятор должен убедиться, что значение, переданное f в качестве первого параметра, не изменилось при оценке второго аргумента. Семантика оценки вызова функции использует приведённый ниже алгоритм (реализации могут отличаться, если они обеспечивают такой же результат).

1. Аргументы оцениваются слева направо в порядке их указания в вызывающем функцию выражении.
2. Для каждого параметра out и inout соответствующее l-значение сохраняется (поэтому оно не может измениться в результате оценки следующих аргументов). Это важно, если аргументы включают операции индексирования в стеке заголовков.
3. Значение каждого аргумента сохраняется во временной области.
4. Функция вызывается со значениями из временной области в качестве аргументов. Мы гарантируем, что сохранённые во временной области значения не являются псевдонимами, поэтому вызов функции может быть реализован с использованием «вызова по ссылке» (call-by-reference), если архитектура позволяет это.
5. При завершении работы функции (возврате) значения из временной области, соответствующие аргументам out и inout, копируются слева направо в l-значения, сохранённые в п. 2.

В соответствии с этим алгоритмом вызов предыдущей функции эквивалентен приведённой ниже последовательности операторов.

```
bit tmp1 = a;           // Оценка a, запись результата
bit tmp2 = g(a);        // Оценка g(a), запись результата, изменение a
f(tmp1, tmp2); // Оценка f, изменение tmp1
a = tmp1;               // Копирование результата inout обратно в a
```

Чтобы понять важность этапа 2 в приведённом выше алгоритме, рассмотрим пример

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

Оценка этого вызова будет эквивалентна приведённой ниже последовательности операторов.

```
bit tmp1 = a;                  // Запись значения 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 - это не s[a].z
```

При использовании объектов extern в качестве аргументов их можно передавать лишь без указания направления (см. пример коммутатора VSS).

6.7.1. Обоснование

Основная причина использования семантики copy-in/copy-out (вместо более распространённой семантики call-by-reference) заключается в контроле побочных эффектов использования внешних функций и методов. Функции и методы extern являются основным механизмом, с помощью которого программа P4 взаимодействует со своим окружением. Семантика copy-in/copy-out не позволяет внешним функциям ссылаться на программные объекты P4 и это позволяет компилятору ограничивать побочные эффекты, которые внешние функции могут создавать для программ P4 в пространстве (нет возможности воздействовать на параметры) и времени (побочные эффекты возможны только во время вызова).

В общем случае возможности внешних функций не ограничены - они могут хранить информацию в глобальном хранилище, создавать отдельные потоки, группироваться для совместного использования информации, но не имеют доступа к переменным внутри программ P4. С помощью семантики copy-in/copy-out компилятор может передавать информацию между программами P4 и функциями extern.

Ниже перечислены дополнительные преимущества, обеспечиваемые семантикой copy-in copy-out.

- Возможность компилировать программы P4 для архитектуры, не поддерживающей ссылок (например, содержащих данные в именованных регистрах). Такая архитектура может требовать доступности индексов стека заголовков, которые появляются в программе, в момент компиляции.
- Упрощение анализа для некоторых компиляторов, поскольку параметры функции никогда не будут псевдонимами друг друга внутри функции.

```
parameterList
        : /* Пусто */
        | nonEmptyParameterList
        ;
nonEmptyParameterList
        : parameter
        | nonEmptyParameterList ',' parameter
        ;
parameter
```

```
        : optAnnotations direction typeRef name
        ;
direction
        : IN
        | OUT
        | INOUT
        | /* Пусто */
        ;
```

Ниже кратко перечислены ограничения, связанные с направлением для параметров.

- При использовании объектов extern в качестве параметров они могут передаваться лишь без направления.
- Все параметры конструктора оцениваются во время компиляции и по этой причине они не могут иметь направления (не могут быть in, out или inout) - это применимо к объектам package, control, parser и extern. Значения этих параметров должны быть заданы во время компиляции и должны оцениваться по известным в момент компиляции значениям (см. раздел 13).
- Для действий все параметры без направления должны размещаться в конце списка параметров. Когда действие появляется в списке действий таблицы, требуется привязка лишь для параметров с направлениям (см. параграф 12.1).
- Действия могут также вызываться явно с использованием синтаксиса вызова функций из блока управления или другого действия. В таких случаях значения всех параметров действия должны быть представлены явно, включая значения параметров без направления. Параметры без направления в этом случае ведут себя подобно параметрам in (см. параграф 12.1.1).

6.8. Преобразование имён

Объекты P4, создающие пространства имён, организованы иерархически. Имеется безымянное пространство верхнего уровня, содержащее все объявления этого уровня.

Идентификаторы, имеющие префикс в виде точки, всегда преобразуются в пространство 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 верхнего уровня bit<32>
        }
}
```

Ссылки при преобразовании идентификаторов применяются в направлении изнутри наружу, начиная с текущей области действия и далее во все лексически охватывающие области. Компилятор может выдавать предупреждение, если для одного имени возможно несколько преобразований (затенение имён).

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

6.9. Видимость

Идентификаторы, определённые в пространстве верхнего уровня, имеют глобальную видимость. Объявления внутри parser или control являются «приватными» и на них невозможны ссылки извне охватывающего анализатора или элемента управления.

7. Типы данных P4

Язык P4₁₆ является статически типизованным. Программы, которые не проходят проверку типов, считаются недействительными и отвергаются компилятором. P4 поддерживает множество базовых типов, а также операторы типа для создания производных типов. Некоторые значения преобразуются в другой тип с помощью cast. Однако для того, чтобы намерения пользователя оставались понятными, неявные преобразования разрешаются лишь в редких случаях, а диапазон доступных преобразований осознанно ограничен.

7.1. Базовые типы

Ниже перечислены базовые типы языка P4.

- Тип void не имеет значений и может использоваться лишь в нескольких случаях.
- Тип error служит для передачи информации об ошибках, определяемой компилятором в зависимости от платформы.
- Тип match_kind служит для описания реализации поиска в таблицах.
- Тип bool представляет логические значения (Boolean)
- Битовые строки фиксированного размера обозначаются bit<>
- Целые числа со знаком фиксированной разрядности представляемые в форме дополнения до двух - int<>.
- Битовые строки с динамически определяемым максимальным размером - varbit<>

```
baseType
        : BOOL
```

```
        | ERROR
        | BIT
        | BIT '<' INTEGER '>'
        | INT '<' INTEGER '>'
        | VARBIT '<' INTEGER '>'
        ;
```

7.1.1. Тип void

Тип void является пустым и не имеет значений. Он не включён в правило baseType, поскольку может появляться лишь в ограниченных случаях.

7.1.2. Тип error

Тип error type содержит коды, которые могут использоваться для сигнализации ошибок. Новые константы типа error определяются с использованием показанного ниже синтаксиса.

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

Все константы типа error помещаются в пространство ошибок, независимо от места определения этой константы. Тип error похож на перечисляемые (enum) типы в других языках. Программа может включать множество объявлений типа error, которые компилятор будет собирать вместе. Если один и тот же идентификатор типа error объявлен в разных местах, это вызывает ошибку. Выражения типа error описаны в параграфе 8.2.

Приведённое ниже объявление создаёт две константы типа error (эти ошибки объявлены в библиотеке ядра P4).

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

Представление ошибок зависит от платформы.

7.1.3. Тип match_kind

Тип match_kind очень похож на тип error и применяется для объявления набора имён, которые могут использоваться в свойствах ключей таблиц (см. параграф 12.2.1). Все идентификаторы помещаются в пространство имён верхнего уровня. Объявление одного и того же идентификатора match_kind несколько раз является ошибкой.

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

Библиотека ядра P4 содержит приведённое ниже объявление

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

Архитектура может поддерживать различные match_kind. Объявления новых match_kinds могут включаться только в файлы описания модели, программисты P4 не могут объявлять новые виды сопоставлений.

7.1.4. Тип bool

Логический тип bool включает два значения - false и true. Логические значения не относятся к integer или bit-string.

7.1.5. Строки

P4 не поддерживает обработку строк. Единственным типом строк, которые могут присутствовать в программах P4, являются строковые константы, описанные в параграфе 6.3.3.3. Эти константы могут применяться лишь в аннотациях (раздел 17). Например, приведённая ниже аннотация указывает, что конкретное имя должно использоваться для таблицы при генерации API уровня управления.

```
@name("acl") table t1 { ...}
```

7.1.6. Целые числа

P4 поддерживает целочисленные значения произвольного размера. Правила типизации для целых чисел выбираются в соответствии с описанными ниже принципами.

- Подобие языку C. Типизация целых чисел основана на подходах языка C, расширенных для работы со значениями произвольного фиксированного размера. В частности, тип результата выражения зависит лишь от операндов этого выражения, а не от способа использования (потребления) этого результата.
- Отсутствие неопределённости в поведении. P4 пытается избежать многих особенностей поведения C, включая размер целочисленных переменных (int), которые могли приводить к переполнению, а также неожиданным результатам для некоторых комбинаций входных данных (например, смещение на отрицательную величину, переполнение для целых чисел со знаком и т. п.). Комбинации целых чисел в P4 не приводят к неопределённому поведению.
- Отсутствие сюрпризов. Правила типизации P4 выбраны так, чтобы поведение программ было максимально близко к поведению хорошо работающих программ C.
- Запрет вместо сюрпризов. В тех случаях, где могли возникать неожиданные результаты (например, сравнение целых чисел со знаком и без знака в C), было принято решение о запрете выражений с неоднозначной интерпретацией. Например, P4 не разрешает двоичные операции с комбинацией целых чисел со знаком и без знака.

Приоритет арифметических операций аналогичен принятому в C (например, умножение выполняется до сложения).

7.1.6.1. Переносимость

Ни одна платформа (target) P4 не может поддерживать все возможные типы и операции. Например, тип bit<23132312> разрешён в P4, но его поддержка в реальных платформах крайне маловероятна. Каждая платформа может вносить ограничения в набор поддерживаемых типов, которые могут включать:

- максимальный поддерживаемый размер;
- требования к выравниванию и заполнению (например, арифметические выражения могут поддерживаться лишь для значений, образованных целым числом байтов);
- ограничения на поддерживаемые операции (например, архитектура может поддерживать умножение только для небольших констант или сдвиг только на небольшие значения).

В документации платформы должны быть чётко заданы такие ограничения и все компиляторы для конкретных платформ должны выдавать сообщения об ошибках при выходе за пределы заданных платформой ограничений. Архитектура может отвергать правильно типизованные программы P4 и оставаться совместимой со спецификацией P4. Однако если архитектура сочла программу P4 пригодной, поведение при работе этой программы должно соответствовать данной спецификации.

7.1.6.2. Целые числа без знака (битовые строки)

Целые числа без знака (которые называют также битовыми строками - bit-string) имеют произвольный размер, указываемый в битах. Битовая строка размера W объявляется как bit<W>. Значение W должно быть известно в момент компиляции (см. параграф 16.1) и оцениваться положительным целым числом.

Биты в bit-string нумеруются от 0 до W-1, бит 0 является младшим, W-1 - старшим.

Например, тип bit<128> означает битовую строку из 128 битов с номерами от 0 до 127, где бит 127 является старшим.

Обозначение bit является сокращением для bit<1>.

Архитектуры P4 могут вносить свои ограничения для типов bit, например, может ограничиваться максимальный размер или некоторые арифметические операции будут возможны лишь для некоторых размером (например, 16, 32 и 64).

Операции, которые могут быть выполнены над целыми числами без знака, описаны в параграфе 8.5.

7.1.6.3. Целые числа со знаком

Целые числа со знаком представляются в форме дополнения до 2. Целое число размером W битов объявляется как int<W>. Значение W должно быть известно в момент компиляции и оцениваться положительным целым числом больше 1.

Биты целого числа нумеруются от 0 до W-1, бит 0 является младшим, бит W-1 - старшим.

Например, int<64> описывает целочисленный тип размером 64 бита, где биты нумеруются от 0 до 63 и бит 63 является старшим (знаком).

Архитектуры P4 могут вносить свои ограничения для целых чисел со знаком, например, может ограничиваться максимальный размер или некоторые арифметические операции будут возможны лишь для некоторых размером (например, 16, 32 и 64).

Операции, которые могут быть выполнены над целыми числами без знака, описаны в параграфе 8.6.

7.1.6.4. Битовые строки с динамическим размером

Некоторые сетевые протоколы используют поля, размер которых можно определить лишь в процессе работы (например, опции IPv4). Для поддержки ограниченных манипуляция такими полями P4 обеспечивает специальный тип битовых строк, размер которых определяется во время работы - varbit.

Тип varbit<W> обозначает битовую строку размером не более W, где значение W должно быть известно во время компиляции. Например, тип varbit<120> означает битовую строку, которая может иметь размер от 0 до 120 битов. Большинство операций, применимых к битовым строкам фиксированного размера (целые числа без знака), не пригодны для динамических битовых строк. Архитектуры P4 могут вносить дополнительные ограничения для типов varbit, например, ограничивая максимальный размер или всегда требуя в процессе работы для таких типов размера, кратного целому числу байтов.

Операции для типа varbit описаны в параграфе 8.8.

7.1.6.5. Целые числа с неограниченной разрядностью

Тип данных с неограниченной разрядностью (infinite-precision) описывает целые числа, размер которых не ограничивается. Этот тип обозначается int.

Этот тип зарезервирован для целочисленных литералов (констант) и выражений, содержащих только константы. Во время выполнения программы P4 значения не могут иметь тип int, в процессе компиляции все значения этого типа преобразуются в типы фиксированного размера в соответствии с описанными ниже правилами.

Операции для типа int описаны в параграфе 8.7.

7.1.6.6. Типы целочисленных констант

Типы целочисленных литералов (констант) перечислены ниже.

- Простые константы типа int.
- Положительные целые размера N с префиксом w типа bit<N>.
- Целые числа размера N с префиксом s типа int<N>.

В таблице приведено несколько примеров целочисленных констант указанных типов. Дополнительные примеры литералов даны в параграфе 6.3.3.

Константа	*Интерпретация*
`10`	Тип int, значение 10
`8w10`	Тип bit<8>, значение 10
`8s10`	Тип int<10>, значение 10
`2s3`	Тип int<2>, значение -1 (2 последних бита), предупреждение о переполнении
`1w10`	Тип bit<1>, значение 0 (последний бит), предупреждение о переполнении
`1s10`	Ошибка - 1-битовый тип по знаком недопустим

7.2. Производные типы

P4 поддерживает множество конструкций, которые можно использовать для создания производных типов:

- enum
- header
- стеки заголовков
- struct
- header_union
- tuple
- специализация типа
- extern
- parser
- control
- package

Типы header, header_union, enum, struct, extern, parser, control и package можно использовать только в объявлениях типов, где они задают новое имя для типа, которое позднее можно применять в качестве идентификатора этого типа.

Другие типы не могут быть объявлены, но синтезируются компилятором для представления типов некоторых конструкций языка. Эти типы описаны в параграфе 7.2.8 и включают set и function. Например, программист не может объявить переменную с типом set, но может создать выражение, которое будет оцениваться как тип set. Эти типы используются при проверке типов.

```
typeDeclaration
        : derivedTypeDeclaration
        | typedefDeclaration
        | parserTypeDeclaration ';'
        | controlTypeDeclaration ';'
        | packageTypeDeclaration ';'
        ;
derivedTypeDeclaration
        : headerTypeDeclaration
        | headerUnionDeclaration
        | structTypeDeclaration
        | headerUnionDeclaration
        | enumDeclaration
        ;
typeRef
        : baseType
        | typeName
        | specializedType
        | headerStackType
        | tupleType
        ;
prefixedType
        : TYPE
        | dotPrefix TYPE
        ;
typeName
        : prefixedType
        ;
```

7.2.1. Перечисляемые типы

Перечисляемый тип enum определяется с помощью показанного ниже синтаксиса.

```
enumDeclaration
        : optAnnotations ENUM name '{' identifierList '}'
        ;
identifierList
        : name
        | identifierList ',' name
        ;
```

Например, объявление

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

создаёт новый перечисляемый тип, содержащий 4 константы (например, Suits.Clubs). Объявление enum создаёт новый идентификатор в текущей области действия для именования созданного типа. Базовое представление таких значений не задаётся, поэтому из «размер» в битах не определён (он зависит от платформы).

Аннотации, представленные нетерминальными optAnnotations, описаны в разделе 17.

Операции над значениями enum описаны в параграфе 8.3.

7.2.2. Типы заголовков

Для объявления типа заголовка служит приведённый ниже синтаксис.

```
headerTypeDeclaration
        : optAnnotations HEADER name '{' structFieldList '}'
        ;
structFieldList
        : /* Пусто */
        | structFieldList structField
        ;
structField
        : optAnnotations typeRef name ';'
        ;
```

где каждый typeRef может указывать битовую строку (фиксированного или переменного размера) или целочисленный тип. Это объявление создаёт новый идентификатор в текущей области действия, имя которого может служить для указания данного типа. Заголовки похожи на структуры в C, содержащие все указанные поля. Однако заголовок дополнительно включает скрытое логическое поле validity. Когда флаг validity имеет значение true, заголовок считается подходящим. При создании заголовка автоматически устанавливается validity = false. Для манипуляций с битом пригодности используются методы типа header - isValid(), setValid() и setInvalid(), описанные в параграфе 8.14.

Типы заголовков могут быть пустыми

```
        header Empty_h { }
```

Отметим, что бит validity имеется и у пустых заголовков.

Заголовки, не включающие полей varbit имеют фиксированный размер, в противном случае размер будет переменным. Размер фиксированного заголовка (в битах) является постоянным и определяется суммой размеров всех включённых в заголовок полей (без учёта бита validity). Выравнивания или заполнения для полей заголовка не используется. Архитектура может задавать дополнительные ограничения для типов заголовков (например, ограничение возможного размера целым числом байтов).

Ниже в качестве примера приведено объявление для типичного заголовка Ethernet.

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

Приведённый ниже фрагмент объявляет переменную нового типа Ethernet_h.

```
        Ethernet_h ethernetHeader;
```

Анализатор P4 обеспечивает способы извлечения полей заголовка из пакетов, как описано в параграфе 11.8. При успешном выполнении операции извлечения заголовков 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 описано в параграфе 8.11, заголовки с полями переменного размера могут потребовать анализа в несколько проходов путём его разбиения на несколько заголовков.

7.2.3. Стеки заголовков

Стек заголовков представляет собой массив заголовков. Определение типа приведено ниже.

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

Здесь typeName указывает имя типа заголовков. Для стека hs[n] параметр n задаёт максимальный размер и должен быть положительны целым числом, известным в момент компиляции. Вложенные стеки заголовков не поддерживаются. Во время выполнения программы стек содержит n значений типа typeName, из который пригодными может быть только часть. Выражения для теков заголовков рассмотрены в параграфе 8.15.

Например, определение

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

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

7.2.4. Объединения заголовков

Объединение заголовков представляет собой варианты разных заголовков, из которых может быть выбрано не более одного. Такое объединение можно использовать для представления опций а протоколах типа TCP и IP. Это также говорит компилятору P4 о присутствии лишь одного варианта и позволяет сэкономить ресурсы хранилища.

Объединение заголовков определяется, как показано ниже.

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

Это объявление вводит в текущей области новый тип с указанным именем. Каждый элемент списка полей, служащий для определения объединения заголовков должен быть типом заголовка, однако список полей может быть пустым.

Например, показанный ниже тип Ip_h представляет объединение заголовков IPv4 и IPv6.

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

Объединение заголовков не считается типом фиксированного размера.

7.2.5. Структурированные типы

Структурные типы P4 определяются с помощью показанного ниже синтаксиса.

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

Это объявление вводит в текущей области новый тип с указанным именем. Список элементов может быть пустым. Например, показанная ниже структура Parsed_headers содержит заголовки, распознанные одним анализатором.

```
header Tcp_h { ... }
header Udp_h { ... }
struct Parsed_headers {
        Ethernet_h     ethernet;
        Ip_h           ip;
        Tcp_h          tcp;
        Udp_h          udp;
}
```

7.2.6. Кортежи

Кортеж (tuple) похож на структуру тем, что содержит множество значений. Однако элементы кортежа не являются именованными полями в отличие от элементов структуры. Тип кортежа с n компонентами типов T1,...,Tn записывается как tuple<T1,...,Tn>

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

Операции над кортежами описаны в параграфах 8.10 и 8.12.

7.2.7. Правила для вложенных типов

В таблице указаны все типы, которые могут присутствовать в header, header_union, struct и tuple. Напомним, что int означает целое число неограниченной разрядности без указания размера.

Тип элемента	Тип контейнера		
	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	ошибка	ошибка	разрешено
enum	ошибка	ошибка	разрешено
header	ошибка	разрешено	разрешено
Стек заголовков	ошибка	ошибка	разрешено
header_union	ошибка	ошибка	разрешено
struct	ошибка	ошибка	разрешено
tuple	ошибка	ошибка	разрешено

Обоснование. Тип int не имеет точного размера в отличие от bit<> и int<>. Значения match_kind бесполезно сохранять в переменных, поскольку они служат лишь для указания, как сопоставлять поля с ключами поиска в таблице, а это выполняется в момент компиляции. Тип void бесполезен в какой-либо структуре данных. Заголовки должны иметь точно определённые форматы (как последовательность битов), чтобы их можно было анализировать и собирать.

Отметим, что метод извлечения с двумя аргументами (параграф 11.8.2) для пакетов поддерживается только в полях заголовка varbit.

7.2.8. Синтезированные типы данных

Для проверки типов компилятор P4 может синтезировать некоторые типы представления, которые не могут быть непосредственно выражены пользователем. Эти типы set и function описаны ниже.

7.2.8.1. Типы set

Тип set<T> описывает множество значений типа T и может присутствовать только в ограниченных случаях контекста программ P4. Например, диапазон 8w5 .. 8w8 описывает множество 8-битовых целых чисел 5, 6, 7 и 8, следовательно его типом будет set<bit<8>>;. Это выражение можно использовать в качестве метки в выбранном выражении (см. параграф 11.6), соответствующей любому значению из диапазона. Типы set не могут быть именованными и программисты P4 не могут объявлять их, поскольку эти типы синтезируются компилятором и используются для проверки типов. Выражения с типами set описаны в параграфе 8.12.

7.2.8.2. Типы function

В настоящее время типы function не могут явно создаваться в программах P4 и синтезируются компилятором для представления типов функций, процедур и методов в процессе проверки типов. Тип функции называется также её сигнатурой. Библиотеки могут содержать определения внешних (extern) функций.

Например, объявление

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

описывает объект random, которые имеет тип function и представляет следующую информацию:

- результат имеет тип void;
- функция имеет два входных параметра;
- первый параметр имеет направление in, тип bit<5> и имя logRange;
- второй параметр имеет направление out, тип bit<32> и имя value.

7.2.9. Внешние типы

P4 поддерживает объявления внешних (extern) объектов и функций с использованием приведённого ниже синтаксиса.

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

7.2.9.1. Внешние функции

Объявление функции extern описывает имя и тип сигнатуры функции, но не её реализацию.

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

Пример объявления функции extern приведён в параграфе 7.2.8.2.

7.2.9.2. Внешние объекты

Объявление внешнего объекта указывает сам объект и все методы, которые могут быть вызваны для выполнения расчётов и изменения состояния объекта. Объявления внешних объектов могут также объявлять методы конструктора, который должен иметь такое же имя как включающий его тип extern и не иметь параметров и типа возврата. Возможность присутствия внешних объектов определяется архитектурной моделью и может зависеть от платформы.

```
methodPrototypes
        : /* Пусто */
        | methodPrototypes methodPrototype
        ;
methodPrototype
        : functionPrototype ';'
        | TYPE '(' parameterList ')' ';' // Конструктор
        ;
typeOrVoid
        : typeRef
        | VOID
        | nonTypeName
        ;      // Может быть переменной типа
optTypeParameters
        : /* Пусто */
        | '<' typeParameterList '>'
        ;
typeParameterList
        : nonTypeName
        | typeParameterList ',' nonTypeName
        ;
```

Например, библиотека ядра P4 определяет два extern-объекта packet_in и packet_out, используемых для манипуляций с пакетами (см. параграф 11.8 и раздел 14). Ниже приведён пример вызова методов этого объекта 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
```

```
        }                               // путём вызова метода emit
}
```

Из числа конструкций P4 только функции и методы поддерживают множественный запуск. В одной области действия может существовать множество методов с одним именем, но даже в этом случае две функции (или метода внешнего объекта) могут использовать одно имя лишь при наличии у них разного числа .

7.2.10. Специализация типа

Базовый тип можно специализировать путём задания аргументов для его переменных типа. В случаях, когда компилятор может вывести типы аргументов, специализация типа не требуется. При специализации типа все его переменные типа должны быть привязаны.

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

Например, приведённое ниже объявление extern описывает базовый блок регистров, где типы элементов, хранящихся в каждом регистре, являются произвольными T.

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

Тип T задаётся при создании экземпляра набора регистров путём указания типа Register

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

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

7.2.11. Типы анализаторов и управляющих блоков

Типы анализаторов и блоков управления похожи на типы функций и описывают сигнатуры анализаторов и блоков управления. Такие функции не возвращают значений. Объявления анализаторов и блоков управления в архитектуре могут быть тазовыми (т. е. включать параметры типа).

Типы parser, control и package не могут служить типами аргументов для методов, анализаторов, элементов управления, таблиц и действий. Они могут применяться в качестве типов аргументов, передаваемых конструкторам (см. раздел 13).

7.2.11.1. Объявление типа анализатора

Объявление типа parser описывает сигнатуру анализатора. Анализатору следует иметь хотя бы один аргумент типа packet_in, представляющий обрабатываемый пакет.

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

Например, приведённое ниже объявление типа анализатора с именем P параметризовано переменной типа H. Анализатор получает в качестве входного packet_in значение b и даёт на выходе два значения:

- значение определённого пользователем типа H;
- значение предопределённого типа Counters.

```
struct Counters { ... }
parser P<H>(packet_in b,
        out H packetHeaders,
        out Counters counters);
```

7.2.11.2. Объявления типа элемента управления

Объявление типа control описывает сигнатуру блока управления.

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

Объявления типа control похожи на объявления типа parser.

7.2.12. Типы программ

Объявление типа package описывает сигнатуру программы.

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

Все параметры программы оцениваются во время компиляции и поэтому они не должны иметь направления (т. е., in, out или inout). В остальном объявление типов package очень похоже на объявление parser. Для программ могут лишь создаваться экземпляры, другого поведения во время работы с ними не связывается.

7.2.13. Типы _

Тип don't care (_) может использоваться в некоторых случаях. Его следует указывать лишь в позициях, где может быть переменная привязанного типа. Подчёркивание можно использовать для упрощения кода в тех случаях, когда привязка переменной типа не важна (при унификации типа этот тип может быть объединён с любым другим типом). Пример использования этого типа приведён в параграфе 15.1.

7.3. typedef

Объявление typedef может служить для задания другого имени имеющегося типа.

```
typedefDeclaration
        : TYPEDEF typeRef name ';'
        | TYPEDEF derivedTypeDeclaration name ';'
        | annotations TYPEDEF typeRef name ';'
        | annotations TYPEDEF derivedTypeDeclaration name ';'
        ;
typedef bit<32> u32;
typedef struct Point { int<32> x; int<32> y; } Pt;
typedef Empty_h[32] HeaderStack;
```

Два типа трактуются как синонимы и все операции, которые могут быть выполнены с исходным типом, применимы также к вновь созданному.

8. Выражения

В этом разделе описаны все выражения, которые могут применяться в P4, сгруппированные по типу результата.

Грамматическое правило создания для выражений общего назначения приведено ниже.

```
expression
        : INTEGER
        | TRUE
        | FALSE
        | STRING_LITERAL
        | nonTypeName
        | '.' nonTypeName
        | expression '[' expression ']'
        | expression '[' expression ':' expression ']'
        | '{' expressionList '}'
        | '(' 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       // проверка непрерывности >>
        | expression LE expression           // LE это «меньше или равно» <=
        | expression GE expression
        | expression '<' expression
        | expression '>' expression
        | expression NE expression           // NE это «не равно» !=
        | expression EQ expression           // EQ это «равно» ==
        | expression '&' expression
        | expression '^' expression
        | expression '|' expression
        | expression PP expression           // PP это «конкатенация» ++
        | expression AND expression   // AND это &&
        | expression OR expression           // OR это ||
        | expression '?' expression ':' expression
        | expression '<' typeArgumentList '>' '(' argumentList ')'
        | expression '(' argumentList ')'
        | typeRef '(' argumentList ')'
        | '(' typeRef ')' expression
        ;
expressionList
        : /* Пусто */
        | expression
        | expressionList ',' expression
        ;
member
        : name
        ;
argumentList
        : /* Пусто */
        | nonEmptyArgList
        ;
nonEmptyArgList
        : argument
        | nonEmptyArgList ',' argument
        ;
argument
        : expression
        ;
typeArg
        : DONTCARE
        | typeRef
        ;
typeArgumentList
        : typeArg
```

```
    | typeArgumentList ',' typeArg
    ;
```

Полная грамматика P4 приведена в Приложении E.

Эта грамматика не показывает приоритет различных операторов, поскольку их порядок в точности соответствует принятому в C. Конкатенация (++) имеет такой же приоритет, как сложение. Битовая выборка a[m:l] имеет такой же приоритет, как индексирование массива (a[i]). Требуется дополнительная семантическая проверка для сдвига вправо, чтобы убедиться в отсутствии пробелов между двумя последовательными символами >>. Это правило требуется для возможности анализа операторов сдвига вправо и специализированных типов, как в function<bit<32>>.

Кроме этих выражений P4 поддерживает также выражения для выбора (параграф 11.6), которые могут применяться только в анализаторах.

8.1. Порядок вычисления выражений

Для композитных выражений порядок оценки отдельных компонент имеет важное значение особенно в тех случаях, когда субвыражения могут давать побочные эффекты. Правила вычисления выражений P4 приведены ниже.

- Логические операции && и || используют метод «короткого замыкания» (short-circuit), т. е. вторая операция выполняется только при необходимости.
- Для условного оператора e1 ? e2 : e3 сначала вычисляется e1, затем e2 или e3.
- Все прочие выражения вычисляются слева направо, как они указаны в тексте программы.
- Оценка вызовов методов и функций выполняется в соответствии с параграфом 6.7.

8.2. Операции над типом error

Символьные имена, объявляемые для типа error относятся к пространству имён ошибок. Для типа error поддерживаются только сравнения в виде равенства (==) и неравенства (!=). Результатом такого сравнения является логическое значение.

Ниже приведён пример проверки наличия ошибок.

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

8.3. Операции над типом enum

Символьные имена, объявляемые для типа enum относятся к пространству имён этого типа, а не к пространству имён верхнего уровня.

```
enum X { v1, v2, v3 }
X.v1     // Ссылка на v1
v1       // Ошибка - v1 не относится к пространству имён верхнего уровня
```

Подобно error, для enum поддерживаются только сравнения в виде равенства (==) и неравенства (!=). Выражения типа enum не могут быть приведены к какому-либо иному типу и наоборот.

Отметим, что при появлении значения типа enum в API уровня управления компилятор должен выбрать подходящий упорядоченный тип данных и представление.

8.4. Логические выражения

Для логических выражений поддерживаются операции И (And) - &&, ИЛИ (Or) - ||, НЕ - !, равно - == и не равно - !=.

Порядок действий аналогичен принятому в C, используется сокращение (short-circuit evaluation).

P4 не делает неявных преобразований логических значений в битовые строки и обратно. Поэтому принятые в C выражения вида

```
if (x) ...
```

(где x имеет целочисленный тип) должны в P4 иметь вид

```
if (x != 0) ...
```

Проверка таких выражений описана в параграфе 8.9.2.

8.4.1. Условный оператор

Условные выражения вида e1 ? e2 : e3 ведут себя так же, как аналогичные выражения C. Как описано выше, сначала оценивается выражение e1, затем, в зависимости от результата, - e2 или e3.

Первое субвыражение e1 должно иметь логический тип, второе и третье субвыражения должны быть одного типа и не могут быть целыми числами неограниченной разрядности, пока само условие не может быть оценено в момент компиляции. Это ограничение задано для того, чтобы обеспечить возможность определения размера результата статически в момент компиляции.

8.5. Операции над типом bit (целые числа без знака)

В этом параграфе описаны операции, которые могут быть выполнены над выражениями типа bit<W> для некого размера W, называемого также битовыми строками.

Арифметические операции выполняются с отбрасыванием старших битов, выходящих за пределы размера, подобно операциям с целыми числами без знака в языке C (т. е., представление значений, выходящих за пределы размера W лишь младшими W битами). В частности, P4 не имеет арифметических исключений - результат арифметической операции определён при всех возможных входных значениях.

Все бинарные операции (кроме сдвигов) требуют, чтобы оба операнда имели одинаковый тип и размер, а отступление от этого правила приведёт к ошибке при компиляции. Компилятор не использует неявного преобразования типов для выравнивания размера. Нет бинарных операций (за исключением сдвига), в которых могли бы использоваться одновременно значения со знаком и без знака. Ниже перечислены операции, поддерживаемые в выражениях с битовыми строками.

- Проверка равенства битовых строк одного размера ==, дающая в результате логическое значение.
- Проверка неравенства битовых строк одного размера !=, дающая в результате логическое значение.
- Сравнение беззнаковых значений <, >, <=, >=, оба операнда которого должны иметь одинаковый размер, дающее в результате логическое значение.

Приведённые ниже операции дают в результате битовую строку, если оба операнда являются битовыми строками одного размера.

- Отрицание (-), результатом которого является результат вычитания значения из 2W. Результат является беззнаковым и имеет такой же размер, как входное значение. Семантика операции совпадает с принятой в языке C для целых чисел без знака.
- Унарный плюс (+) - эта операции эквивалентна отсутствию операций.
- Сложение (+). Ассоциативная операция, результат которой рассчитывается путём отсечения старших битов суммы, выходящих за пределы размера строки битов (подобно C).
- Вычитание (-). Результат операции не имеет знака и его тип совпадает с типом операндов. Значение определяется сложением первого операнда с отрицанием второго (подобно C).
- Умножение (*). Размер результата совпадает с размерами операндов и вычисляется путём отбрасывания старших битов произведения, выходящих за пределы размера. Архитектуры P4 могут вносить дополнительные ограничения (например, возможность умножения только на степени 2).
- Побитовая операция И для операндов одного размера (&).
- Побитовая операция ИЛИ для операндов одного размера (|).
- Побитовое «дополнение» для одной битовой строки (~).
- Побитовая операция Исключительное-ИЛИ (XOR) для операндов одного размера (^).

Для битовых строк также поддерживаются перечисленные ниже операции.

- Конкатенация двух битовых строк, обозначаемая ++. Результатом является битовая строка, размер которой равен сумме размеров операндов, а старшие биты принимаются из левого (первого) операнда.
- Извлечение набора последовательных битов, называемого также «выборкой» (slice), обозначаемое [m:l], где m и l - положительные целые числа, которые известны во время компиляции и m >= l. Результатом является битовая строка размером m - l + 1, включающая биты с номерами от l (младший бит результата) до m (старший бит результата) из исходного операнда. Условия 0 <= l < W и l <= m < W проверяются статически (W - размер исходной битовой строки). Отметим, что обе конечные точки включаются в выборку. Границы выборки должны быть известны в момент компиляции, чтобы размер результата можно было определить во время компиляции. Выборки также являются l-значениями, что говорит о поддержке в P4 операторов присваивания e[m:l] = x . Результатом этого выражения является установка битов от m от l в битовой строке e в соответствии с битами x и сохранение неизменными остальных битов e.
- Логический сдвиг влево и вправо на известное во время работы значение, обозначаемый << и >>, соответственно. В операциях сдвига левый операнд является целым числом без знака, а правых должен быть выражением типа bit<S> или неотрицательной целочисленной константой. Тип результата совпадает с типом левого операнда. Сдвиг на величину, превышающую размер левого операнда, приводит к установке 0 во всех битах результата.

8.6. Операции над целыми числами фиксированного размера со знаком

В этом параграфе описаны все операции, которые могут быть выполнены в выражениях типа int<W> для некого размера W. Напомним, что int<W> означает целое число со знаком размером W битов, представленное в форме дополнения до 2.

В общем случае арифметические операции P4 не обнаруживают «опустошения» (underflow) или переполнения (overflow) и просто «циклическая арифметика» (wrap around), подобно операциям с целыми числами без знака в языке C. Поэтому попытка представить значение, размер которого выходит за пределы W, будет приводить к потере старших битов сверх W.

P4 также не поддерживает арифметических исключений и во время работы результат арифметической операции определён для всех комбинаций входных значений.

Все бинарные операции (кроме сдвигов) требуют, чтобы оба операнда имели одинаковый тип и размер, а отступление от этого правила приведёт к ошибке при компиляции. Компилятор не использует неявного преобразования типов для выравнивания размера. Нет двоичных операций (за исключением сдвига), в которых могли бы использоваться одновременно значения со знаком и без знака.

Отметим, что побитовые операции над целыми числами со знаком, определены корректно, поскольку представление определяется дополнением до 2.

Тип int<W> поддерживает перечисленные ниже операции и для всех бинарных операций оба операнда должны быть одного типа. Результат во всех случаях будет иметь результат, размер которого совпадает с размером левого операнда.

- Отрицание, обозначаемое -.
- Унарный плюс (+) - эта операции эквивалентна отсутствию операций.
- Сложение (+).
- Вычитание (-).
- Проверка равенства и неравенства (== и !=, соответственно), дающая в результате логическое значение.
- Численное сравнение <, <=, >, и >=, дающее в результате логическое значение.
- Умножение (*). Размер результата совпадает с размерами операндов и вычисляется путём отбрасывания старших битов произведения, выходящих за пределы размера. Архитектуры P4 могут вносить дополнительные ограничения (например, возможность умножения только на степени 2).
- Арифметический влево и вправо (<< и >>, соответственно). Левый операнд имеет знак, а правый должен быть целым числом без знака типа bit<S> или неотрицательной целочисленной константой. Тип результата совпадает с типом левого операнда. Сдвиг на величину, превышающую размер левого операнда, приводит к установке 0 во всех битах результата.

8.6.1. Замечания о сдвигах

Сдвиг (для значений со знаком и без знака) требует специального рассмотрения по приведённым ниже причинам.

- Сдвиг вправо ведёт себя по-разному для значений со знаком и без знака - в первом случае он является арифметическим.
- Сдвиг на отрицательное значение не имеет чёткой семантики - система типов P4 не допускает сдвига на отрицательные значения.
- В отличие от C сдвиг на величину, превышающую размер операнда, даёт определённый результат.
- В зависимости от возможностей платформы операция сдвига может выполнять работу, которая является возведением в степень числа битов правого операнда.

Рассмотрим пример

```
bit<8> x;
bit<16> y;
... y << x ...
... y << 1024 ...
```

Как отмечено выше, P4 даёт определённый результат при сдвиге на значение, превышающее размер левого операнда (в отличие от C).

Платформы P4 могут вносить дополнительные ограничения на операции сдвига типа запрета сдвига на величину, определяемую не константой (выражение с переменным результатом), или на результат выражения, выходящий за некоторые границы. Например, платформа может запрещать сдвиг 8-битового значения на непостоянную величину, размер которой превышает 3 бита.

8.7. Операции над целыми числами произвольной разрядности

Тип int означает целые числа произвольной разрядности. В P4 все выражения типа int должны иметь известные в момент компиляции значения. Ниже перечислены операции для типа int.

- Отрицание, обозначаемое -.
- Унарный плюс (+) - эта операции эквивалентна отсутствию операций.
- Сложение (+).
- Вычитание (-).
- Проверка равенства и неравенства (== и !=, соответственно), дающая в результате логическое значение.
- Численное сравнение <, <=, >, и >=, дающее в результате логическое значение.
- Умножение (*).
- Целочисленное деление для положительных значений с отбрасыванием остатка (/).
- Деление по модулю для положительных значений (%).
- Арифметический влево и вправо (<< и >>, соответственно). Результат имеет тип int. Правый операнд должен быть положительным. Выражение a << b эквивалентно $a \times 2b$, а a >> b - $\lfloor a/2b \rfloor$.

Все операнды, участвующие в этих операциях, должны иметь тип int. Ни одну из этих операций, за исключением сдвига, нельзя использовать для комбинирования значений типа int со значениями фиксированного размера. Однако компилятор автоматически будет добавлять приведение типов для преобразования int в тип фиксированного размера в случаях, описанных в параграфе 8.9.

Все вычисления значений int выполняются без потери информации. Например, перемножение двух 1024-битовых значений может давать 2048-битовый результат (отметим, что конкретное представление значений int не задаётся). Значения int могут быть приведены к типам bit<w> и int<w>. Приведение значений int к типу с фиксированным размером будет сохранять младшие биты. Если отсечка по размеру будет приводить к потере старших битов, компилятор должен выдавать предупреждение.

Примечание. Побитовые операции (|, &, ^, ~) не определены для выражений типа int. Кроме того, недопустимо использовать деление и деление по модулю для отрицательных значений.

8.8. Операции над битовыми типами переменного размера

Для поддержки анализа заголовков с полями переменного размера в P4 служит тип varbit. Каждое включение типа varbit объявляется статически с максимальным размером, а реальный размер может динамически меняться в заданных пределах. Перед инициализацией битовой строки переменного размера её размер не известен.

Для битовых строк переменного размера поддерживается ограниченный набор операций.

- Извлечение анализатором данных в строку переменного размера с помощью двухаргументного метода внешнего объекта packet_in extern (см. параграф 11.8.3). Эта операция устанавливает динамический размер поля.
- Передача (назначение) значения в другую битовую строку переменного размера. Целевая переменная должна иметь иметь такой же статический размер как исходная. При выполнении операции динамический размер целевой переменной устанавливается в соответствии с динамическим размером источника.
- Метод emit внешнего объекта packet_out, который вставляет битовую строку переменного размера с известным динамическим размером в создаваемый пакет (см. раздел 14).

8.9. Приведение типов

P4 поддерживает ограниченный набор преобразований (приведения) типов (cast). Приведение типа записывается в форме (t) e, где t указывает тип, а e - выражение. Приведение разрешено лишь между базовыми типами. Это создаёт некоторые проблемы для программистов, но и обеспечивает некоторые преимущества:

- делает намерения пользователя однозначными;
- делает явными приведения типов связанные с преобразованием числовых значений (реализация некоторых преобразований включает знаки и это требует на некоторых платформах значительных ресурсов);
- уменьшает число вариантов, принимаемых во внимание спецификацией P4 (некоторые платформы могут поддерживать не все преобразования типов).

8.9.1. Явные преобразования

Перечисленные ниже приведения типов являются допустимыми в P4.

- bit<1> <-> bool - преобразует 0 в false, 1 в true и обратно.
- int<W> -> bit<W> - сохраняет все биты неизменными, интерпретируя отрицательные значения как положительные.
- bit<W> -> int<W> - сохраняет все биты неизменными, интерпретируя значения с установленным старшим битом как отрицательные.
- bit<W> -> bit<X> - отсекает старшие биты, если W > X и заполняет их нулями в противном случае (W <= X).
- int<W> -> int<X> - отсекает старшие биты, если W > X и перемещает бит знака в противном случае (W < X).
- int -> bit<W> - преобразует целочисленное значение в достаточно большую битовую строку с дополнением до 2 для предотвращения потери информации, а затем отсекает старшие биты до размера W. Компилятору следует выдавать предупреждение при переполнении или преобразовании отрицательного значения.
- int -> int<W> - преобразует целочисленное значение в достаточно большую битовую строку с дополнением до 2 для предотвращения потери информации, а затем отсекает старшие биты до размера W. Компилятору следует выдавать предупреждение при переполнении
- Преобразования между двумя типами, созданными typedef, эквивалентные одной из приведённых выше комбинаций.

8.9.2. Неявные преобразования

Для сохранения простоты языка и предотвращения скрытых издержек в P4 неявное приведение типа используется лишь для типов с фиксированной шириной. В частности применение бинарной операции к выражениям типа int и типа с фиксированным размером будут неявно приводить тип int к типу второго выражения.

Например для выражений

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

компилятор будет добавлять неявные приведения типа:

- x + 1 становится x + (bit<8>)1;
- z < 0 становится z < (int<8>)0;
- x << 13 становится 0; предупреждение о переполнении;
- x | 0xFFF становится x | (bit<8>)0xFFF; предупреждение о переполнении.

8.9.3. Недопустимые арифметические выражения

Многие арифметические выражения, разрешённые в других языках, будут недопустимыми в P4. Для иллюстрации рассмотрим следующие выражения:

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

В таблице показано несколько выражений, которые не допускаются правилами типизации P4. Для каждого из этих выражений представлены варианты допустимой записи. Важно отметить, что для некоторых имеется несколько пригодных вариантов, которые могут давать разные результаты! Компилятор не может знать намерений пользователя, поэтому P4 требует от пользователя однозначного указания.

Выражение	*В чем ошибка*	*Варианты*
`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`	Нельзя 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`	RHS для сдвига не может иметь знака	`x << (bit<8>)z`
`x < z`	Разные знаки	`X < (bit<8>)z` `(int<8>)x < z`
`1 << x`	Размер 1 не известен	`32w1 << x`
`~1`	Битовая операция с целым числом	`~32w1`
`5 & -3`	Битовая операция с целым числом	`32w5 & -3`

8.10. Операции над кортежами

Значения типа «кортеж» (tuple) могут назначаться другим кортежам того же типа, передаваться в качестве аргументов функции или возвращаться функцией, а также могут инициализироваться списком выражений.

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

8.11. Операции над списками

Список задаётся с использованием фигурных скобок, элементы разделяются запятыми.

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

Тип списка - кортеж (параграф 7.2.8). Списки могут назначаться в качестве значений выражениям типа tuple, struct или header, а также передаваться методам в качестве аргументов. Списки могут быть вложенными. Однако списки не являются l-значениями.

Например, в приведённом ниже фрагменте используется выражение-список для передачи нескольких полей заголовков одновременно внешней функции LearningProvider

```
extern LearningProvider {
        void learn<T>(in T data);
}
LearningProvider() lp;
lp.learn( { hdr.ethernet.srcAddr, hdr.ipv4.src } );
```

Список можно использовать для инициализации структуры, если список имеет количество элементов, совпадающее с числом полей структуры. Результатом будет присвоение значения i-го элемента списку i-му полю структуры.

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

Списки можно также использовать для инициализации переменных типа tuple.

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

8.12. Операции над множествами

Некоторые выражения P4 обозначают наборы значений (set<T> для некого типа T, см. параграф 7.2.8.1). Такие выражения могут появляться лишь в некоторых анализаторах контекста и константах в таблицах. Например, выражение select (см. параграф 11.6) имеет показанную ниже структуру.

```
select (expression) {
        set1: state1;
        set2: state2;
        ...
}
```

Здесь выражения set1, set2 и т. п. оцениваются набором значений и select проверяет, относится ли значение к множествам, использованным в качестве меток.

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

```
        | simpleExpressionList ',' simpleKeysetExpression
        ;
simpleKeysetExpression
        : expression
        | DEFAULT
        | DONTCARE
        | expression MASK expression
        | expression RANGE expression
        ;
```

Операторы маскирования (&&&) и диапазона (..) имеют одинаковый приоритет, который непосредственно выше &.

8.12.1. Одноэлементные множества

В контексте множества выражение означает множество с одним элементом. Например, в приведённом ниже фрагменте программы

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

Метка 4 означает одноэлементное множество, содержащее значение 4.

8.12.2. Универсальное множество

В контексте множества выражения default или _ обозначают универсальное множество, содержащее все возможные значения данного типа.

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

8.12.3. Маски

Инфиксный оператор &&& принимает два аргумента типа bit<W> и создаёт значение типа set<bit<W>>. Правый операнд используется в качестве «маски», в которой нулевое значение бита указывает, что соответствующий бит первого операнда не имеет значения (don't care). Формально множество a &&& b определяется как

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

Например,

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

означает множество, содержащее 16 различных 8-битовых значений, имеющих битовый шаблон XXXX1010, где X указывает любое значение бита. Отметим, что возможно множество способов выразить набор ключей с использованием оператора маскирования (например, 8w0xFA &&& 8w0x0F означает то же, что и выражение выше).

Архитектуры P4 могут вносить дополнительные ограничения для выражений в левой и правой части оператора маскирования (например, может требоваться, чтобы одно или оба субвыражения имели известные во время компиляции значения).

8.12.4. Диапазоны

Инфиксный оператор .. принимает два аргумента одного типа T, где T может быть bit<W> или int<W> и даёт в результате значение типа set<T>. Множество содержит все значения из числового интервала между первым и вторым операндом, включительно. Например,

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

означает набор значений 4w5, 4w6, 4w7 и 4w8.

8.12.5. Произведения

Несколько множеств можно объединить 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 будет множество кортежей (tuple).

8.13. Операции над структурированными типами

Единственной операцией для выражений типа struct является доступ к полям, указываемым с помощью точки (.) типа s.field. Если s - l-значение, то s.field тоже будет l-значением. P4 также позволяет копировать структуры с использованием операций присваивания, где источник и назначение имеют одинаковый тип. Структуры могут также инициализироваться списочным выражением, как описано в параграфе 8.11.

8.14. Операции над заголовками

Для заголовков поддерживаются такие же операции, как для структур. При операциях присваивания между заголовками копируется также бит validity.

Кроме того, для заголовков поддерживаются перечисленные ниже методы.

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

Результат чтения или записи поля в недействительном заголовке будет неопределённым. Результат чтения неинициализированного заголовка будет неопределённым. даже в случае пригодного заголовка.

Объект header может быть инициализирован списочным выражением, подобно struct - значения полей списка присваиваются полям заголовка в порядке их указания. В этом случае заголовок автоматически становится действительным.

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

8.15. Операции над стеками заголовков

Стек заголовков представляет собой массив фиксированного размера из заголовков одного типа. Подходящие элементы стека заголовков не обязаны быть непрерывными. P4 обеспечивает набор расчётов для манипуляций со стеками заголовков. Стек hs типа h[n] можно рассматривать в терминах приведённого ниже псевдокода.

```
// Объявление типа
struct hs_t {
        bit<32> nextIndex;
        bit<32> size;
        h[n] data;              // Обычный массив
}
// Экземпляр объявления и инициализация
hs_t hs;
hs.nextIndex = 0;
hs.size = n;
```

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

Для стека заголовков hs размером n приведённые ниже выражения являются допустимыми.

- hs[index] указывает ссылку на заголовок в указанной позиции стека. Если hs является l-значением, результат также будет l-значением. Заголовок может быть непригодным. Некоторые архитектуры могут требовать, чтобы значение выражения index было известно во время компиляции. Обращение к стеку заголовков с индексом меньше 0 или больше hs.size приводит к неопределённому результату.
- hs.size указывает 32-битовое целое число без знака, которое возвращает размер стека заголовков (постоянная величина в момент компиляции).
- Присваивание заголовка из стека hs в заголовок другого стека с такими же типами и размерами. Копируются все компоненты hs, включая элементы и биты validity, а также nextIndex.

Чтобы помочь программистам в создании анализаторов стеков, P4 также обеспечивает расчёты, автоматически выполняемые при разборе элементов стека.

- hs.next указывает ссылку на элемент с индексом hs.nextIndex в стеке и может использоваться только в заголовках. Если счётчик nextIndex не меньше размера стека, оценка выражения приводит к отказу и установке ошибки в error.StackOutOfBounds. Если hs является l-значением, hs.next тоже будет l-значением.
- hs.last указывает ссылку на элемент с индексом hs.nextIndex - 1 в стеке, если такой элемент существует и может использоваться только в анализаторах. Если счётчик nextIndex меньше 1 или больше размера стека, оценка выражения приводит к отказу и установке ошибки в error.StackOutOfBounds. Если hs является l-значением, hs.next тоже будет l-значением.
- hs.lastIndex - 32-битовое целое число без знака, которое представляет индекс hs.nextIndex - 1 и может использоваться только в анализаторах. Если счётчик nextIndex = 0, оценка выражения даёт неопределённое значение.

Кроме того, P4 обеспечивает приведённые ниже расчёты, которые могут быть использованы для манипулирования элементами на вершине и дне стека.

- hs.push_front(int count) сдвигает hs «вправо» на величину count. Первые count элементов становятся недействительными. Последние count элементов стека отбрасываются. Счётчик hs.nextIndex инкрементируется на величину count. Аргумент count должен быть положительным целым числом, значение которого известно в момент компиляции. Возвращаемое значение имеет тип void.
- hs.pop_front(int count) сдвигает hs «влево» на величину count (т. е. элемент с индексом count перемещается в стеке в позицию 0). Последние count элементов становятся недействительными. Счётчик hs.nextIndex декрементируется на величину count. Аргумент count должен быть положительным целым числом, значение которого известно в момент компиляции. Возвращаемое значение 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;
        // Примечание. В this.last, this.next и this.lastIndex устанавливается 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;
	}
	// Примечание. В this.last, this.next и this.lastIndex устанавливается this.nextIndex
}
```

8.16. Операции над объединениями заголовков

Переменная, объявленная с типом union изначально непригодна. Например,

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

Это также означает, что заголовки от h1 до hn в объединении заголовков также непригодны. В отличие от заголовков объединение не инициализируется. Однако объединение заголовков можно сделать пригодным, присваивая действительный заголовок одному из элементов объединения.

```
U u;
H1 my_h1 = { 8w0 };   // my_h1 пригодно
u.h1 = my_h1;         // u и u.h1 не пригодны
```

Можно также присвоить список элементу объединения заголовков.

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

или установить в них биты validity напрямую.

```
U u;
u.h1.setValid();      // u и u.h1 пригодны
H1 my_h1 = u.h1;      // my_h1 пригодно, но содержит неопределённое значение
```

Отметим, что считывание неинициализированного заголовка даёт неопределённый результат даже в тех случаях, когда сам заголовок действителен.

Более формально, если u - выражение, типом которого служит объединение заголовков U с полями, упорядоченными по hi, для манипуляций u можно использовать перечисленные ниже операции.

- u.hi.setValid() устанавливает validity = true в заголовке hi и false - в остальных заголовках (в результате считывание этих заголовков будет давать неопределённое значение).
- u.hi.setInvalid() устанавливает validity = false в любом элементе u, если ранее там было установлено значение true и это приводит к тому, что считывание любого заголовка из u будет давать неопределённый результат.

Можно воспринимать присваивание заголовку

```
u.hi = e
```

как эквивалент

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

если e действительно и

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

в противном случае.

Присваивания между переменными одного типа объединения заголовков разрешены. Присваивание u1 = u2 полностью копирует состояние объединения u2 в u1. Если u2 действительно, в нем присутствует некий заголовок u2.hi, который является действительным. Присваивание ведёт себя так же, как u1.hi = u2.hi. Если u2 не действительно, u1 также становится недействительным (т. е., все действительные заголовки u1 становятся недействительными).

Метод u.isValid() возвращает true, если любой элемент объединения заголовков u является действительным. В противном случае возвращается false. Методы setValid() и setInvalid() не определены для объединений заголовков.

Представление выражения типа объединения заголовков на выдачу (to emit) просто приводит к выдаче одного действительного заголовка, если такой имеется.

Приведённый ниже пример показывает, как можно использовать объединение заголовков для однотипного представления заголовков 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;
        }
}
```

В другом примере объединение заголовков используется для анализа (отдельных) опций TCP.

```
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<32> n = (bit<32>)b.lookahead<Tcp_option_sack_top>().length;
                b.extract(vec.next.sack, n);
                transition start;
        }
}
```

8.17. Вызовы методов и функций

Для вызова методов и функций используется показанный ниже стандартный синтаксис.

```
expression
        : ...
        | expression '<' typeArgumentList '>' '(' argumentList ')'
        | expression '(' argumentList ')'
argumentList
        : /* Пусто */
        | nonEmptyArgList
        ;
nonEmptyArgList
        : argument
        | nonEmptyArgList ',' argument
        ;
argument
        : expression
        ;
typeArgumentList
        : typeRef
        | typeArgumentList ',' typeRef
        ;
```

Аргументы функций оцениваются по порядку слева направо перед реальным вызовом функции. При вызовах используются соглашения copy-in/copy-out (параграф 6.7). Для базовых функций аргументы типа могут быть явно заданы при вызове. Компилятор не добавляет неявных преобразований типа для аргументов методов или функций, типы этих аргументов должны точно соответствовать типам параметров.

При использовании функции в качестве оператора возвращённый функцией результат отбрасывается.

8.18. Вызовы конструкторов

Несколько конструкций P4 обозначают ресурсы, выделяемые во время компиляции:

- внешние объекты;
- анализаторы;
- блоки управления;
- программы.

Выделение таких объектов может выполняться двумя способами:

- с помощью вызова конструктора, который представляет собой выражение, возвращающее объект соответствующего типа;
- путём создания экземпляра, как описано в параграфе 9.3.

Синтаксис вызова конструктора похож на вызов функции. Конструкторы целиком оцениваются в момент компиляции (см. параграф 16). Поэтому все аргументы конструктора должны быть выражениями, которые могут быть оценены во время компиляции.

Приведённый ниже пример показывает вызов конструктора для установки зависящего от платформы свойства таблицы.

```
extern ActionProfile {
       ActionProfile(bit<32> size);  // Конструктор
}
table tbl {
       actions = { ... }
       implementation = ActionProfile(1024);        // Вызов конструктора
}
```

9.Объявления констант и переменных

9.1. Константы

Синтаксис определения постоянных значений (констант) приведён ниже.

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

Такое определение вводит константу со значением заданного типа. Ниже приведены примеры корректного определения констант.

```
const bit<32> COUNTER = 32w0x0;
struct Version {
       bit<32> major;
       bit<32> minor;
}
const Version version = { 32w0, 32w0 };
```

Значение (инициализатор) константы должно быть известно в момент компиляции.

9.2. Переменные

Локальные переменные объявляются с указанием типа и имени, а также может указываться инициализатор (начальное значение) и аннотация.

```
variableDeclaration
        : annotations typeRef name optInitializer ';'
        | typeRef name optInitializer ';'
        ;
optInitializer
        : /* Пусто */
        | '=' initializer
        ;
```

Переменные, объявленные без инициализатора, являются неинициализированными (за исключением стеков заголовков, в которых счётчик nextIndex инициализируется значением 0, как описано в параграфе 8.15). Язык вносит некоторые ограничения на типы переменных - можно использовать большинство типов P4, которые могут быть записаны явно (например, базовые типы, struct, header, header_stack, tuple). Однако невозможно объявить переменные типов, которые могут лишь синтезироваться компилятором (например, set). Кроме того, переменные типа parser, control, package и extern должны декларироваться с помощью создания экземпляров (см. параграф 9.3).

Считывание значения переменной, которая не была инициализирована, даёт неопределённый результат. Компилятору следует следует пытаться обнаруживать такие события и выдавать предупреждения. Объявления переменных могут встречаться в разных местах программ P4:

- в операторах блоков;
- в состоянии parser;
- в теле action;
- в блоке применения блока control;
- в списке локальных объявлений parser;
- в списке локальных объявлений control.

Переменные имеют локальную область действия и ведут себя подобно связанным со стеком переменным в языках типа C. Значение переменной никогда не сохраняется между двумя вызовами включающего её блока. В частности, переменные не могут служить для передачи состояния между разными сетевыми пакетами.

9.3. Создание экземпляров

Создание экземпляра похоже на объявление переменной, но зарезервировано для типов с конструктором (объекты extern, блоки управления, parser, package):

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

Создание экземпляра записывается как вызов конструктора, за которым следует имя. Создание экземпляров всегда происходит во время компиляции (параграф 16.1). В результате создаётся объект с указанным именем, которое привязывается к результату вызова конструктора.

Например, гипотетический набор объектов Counter может быть создан с помощью приведённого ниже фрагмента.

```
// Из библиотеки платформы
enum CounterType {
        Packets,
        Bytes,
        Both
}
extern Counter {
        Counter(bit<32> size, CounterType type);
        void increment(in bit<32> index);
}
// Пользовательская программа
control c(...) {
        Counter(32w1024, CounterType.Both) ctr;     // Создание экземпляра
        apply { ... }
}
```

9.3.1. Ограничения на создание экземпляров верхнего уровня

Программа P4 не может инициализировать элементы управления и анализаторы на верхнем уровне программы (package). Это ограничение предназначено для того, чтобы большинство состояний относилось к самой архитектуре или было локальным для анализатора или элемента управления. Например, приведённая ниже программа не будет корректной

```
// Программа
control c(...) { ... }
c() c1; // Недопустимое создание экземпляра верхнего уровня
```

поскольку экземпляр элемента управления c1 создаётся на верхнем уровне. Отметим, что на верхнем уровне разрешены объявления констант и создание экземпляров внешних объектов.

10. Операторы

Каждый оператор P4 (за исключением операторов блока) должен завершаться символом ; (точка с запятой). Операторы могут появляться в нескольких местах программ:

- внутри состояний parser;
- внутри блока control;
- внутри action.

При этом существуют ограничения на типы операторов, которые могут присутствовать в каждом из таких мест. Например, условные операторы не могут присутствовать в анализаторах, а операторы switch поддерживаются только в блоках управления. Ниже представлен наиболее общий случай для блоков управления.

```
statement
        : assignmentOrMethodCallStatement
        | conditionalStatement
        | emptyStatement
        | blockStatement
        | exitStatement
        | returnStatement
        | switchStatement
        ;
assignmentOrMethodCallStatement
        : lvalue '(' argumentList ')' ';'
        | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
        | lvalue '=' expression ';'
        ;
```

Кроме того, анализаторы поддерживают оператор transition (параграф 11.5).

10.1. Оператор присваивания

Оператор присваивания обозначается знаком равенства (=) и сначала его левая часть оценивается в l-значение, затем правая оценивается в значение, которое после этого копируется в l-value. Производные типы (например, структуры) копируются рекурсивно, копируются все компоненты заголовков, включая биты validity. Присваивание не определено для внешних значений.

10.2. Пустой оператор

Пустой оператор указывается одним символом ; (точка с запятой) и означает отсутствие операций.

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

10.3. Оператор блока

Оператор блока обозначается фигурными скобками и включает группу операторов и объявлений, которые выполняются последовательно. Переменные, константы и созданные экземпляры видны только внутри этого блока.

```
blockStatement
        : optAnnotations '{' statOrDeclList '}'
        ;
statOrDeclList
        : /* Пусто */
        | statOrDeclList statementOrDeclaration
        ;
statementOrDeclaration
        : variableDeclaration
        | constantDeclaration
        | statement
        | instantiation
        ;
```

10.4. Оператор возврата

Оператор возврата return незамедлительно прерывает исполнение включающего его действия или элемента управления. Операторы return не допускаются внутри parser.

```
returnStatement
        : RETURN ';'
        ;
```

10.5. Оператор выхода

Оператор выхода exit незамедлительно прерывает исполнение всех выполняющихся в данный момент блоков - текущего действия (при вызове из action), текущего элемента управления (control) и всех их вызовов. Операторы exit не допускаются внутри parser.

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

10.6. Оператор условия

Условный оператор использует стандартный синтаксис и семантику, похожие на другие языки программирования. Однако условное выражение в P4 должно иметь логический тип (а не целочисленный). Условные операторы не допускаются внутри parser.

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

При использовании вложенных операторов if субоператор else относится в внешнему if, у которого нет else.

10.7. Оператор вариантов

Оператор switch можно применять только в блоках управления.

```
switchStatement
        : SWITCH '(' expression ')' '{' switchCases '}'
        ;
switchCases
        : /* Пусто */
        | switchCases switchCase
        ;
switchCase
        : switchLabel ':' blockStatement
        | switchLabel ':'      // Пройти к следующему варианту.
        ;
switchLabel
        : name
        | DEFAULT
        ;
```

Выражения в операторах switch ограничены теми, результатом которых является вызов таблицы (см. параграф 12.2.2).

Если за меткой в операторе switch не следует оператор блока, выполняется переход к следующей метке, а при наличии оператора блоков он выполняется без перехода к следующей метке. Отметим, что это отличается от языка C, где требуется оператор break для предотвращения перехода к следующему варианту. Допускается отсутствие в некоторых действиях какого-либо совпадающего варианта, а также отсутствие используемой по умолчанию метки. Если во время работы соответствующего варианта не найдено, просто продолжится выполнение программы. Следует отметить, что не допускается включение одной и той же метки несколько раз в одном операторе switch.

```
switch (t.apply().action_run) {
        action1:        // Переход к следующему варианту (action2:).
        action2: { ...}
        action3: { ...}
        // В action2 - action3 нет переходов к следующей метке.
}
```

Отметим, что принятая по умолчанию метка оператора switch применяется независимо от того, было ли найдено соответствие. Принятая по умолчанию метка не означает отсутствия (table miss) и выполняется default_action.

11. Анализ пакета

В этом разделе описаны конструкции P4, относящиеся к анализу сетевых пакетов.

11.1. Состояния анализатора

Анализатор P4 описывает машину состояний с одним стартовым и двумя финальными состояниями. Стартовое состояние называется start, финальные - accept (успешный анализ) и reject (отказ при анализе). Состояние start является частью анализатора, а состояния accept и reject не задаются пользователем и логически не являются частями анализатора. На рисунке 8 представлена общая структура машины состояний синтаксического анализатора.

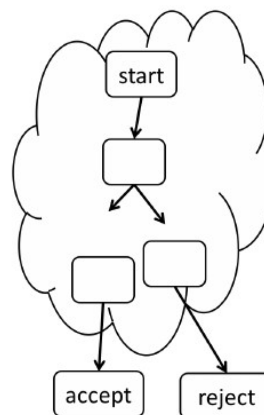

Рисунок 8. Структура FSM анализатора.

11.2. Объявления анализаторов

Объявление анализатора состоит из имени, списка параметров, необязательного списка параметров конструктора, локальных элементов и состояний анализатора (а также необязательных аннотаций).

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

```
parserStates
        : parserState
        | parserStates parserState
        ;
```

Описание параметров конструктора optConstructorParameters, которые полезны для создания параметризуемых анализаторов, приведено в разделе 13.

В отличие от объявлений типа parser объявления анализаторов не могут быть базовыми (generic), т. е. приведённое ниже определение является недопустимым.

```
parser P<H>(inout H data) { ... }
```

Поэтому используемому в контексте parserDeclaration правилу parserTypeDeclaration не следует приводить параметры типа.

В анализаторе должно присутствовать по крайней мере одно состояние - start. Анализатор не может определять два состояния с одним именем. Недопустимо также явное определение состояний accept и reject, поскольку эти состояния логически отделены от состояний анализатора, определяемых программистом.

Объявления состояний описаны ниже. При обработке состояний анализатор может также включать список локальных элементов, каковыми могут быть константы, переменные и экземпляры объектов, которые могут применяться внутри анализатора. Такие объекты могут быть экземплярами внешних объектов или другими анализаторами, вызываемыми как подпрограммы. Однако недопустимо создавать внутри анализатора экземпляр блока управления.

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

Пример полного объявления анализатора представлен в параграфе 5.3.

11.3. Абстрактная машина анализа

Семантику анализатора P4 можно описать в терминах абстрактной машины, которая манипулирует структурой данных ParserModel. В этом параграфе абстрактная машина описана с использованием псевдокода.

Анализатор начинает работу в состоянии start и завершает процесс при достижении состояния reject или accept.

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

Архитектура должна задавать поведение при переходе анализатора в состояние accept и reject. Например, архитектура может задать для всех пакетов, достигших состояния reject, отбрасывание без дальнейшей обработки. Другим вариантом может служить передача таких пакетов в блок, следующий за анализатором, вместе с внутренними метаданными, показывающими, что анализатор достиг состояния reject и информацией об ошибке.

11.4. Состояния анализатора

Состояние анализатора объявляется с использованием приведённого ниже синтаксиса.

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

Каждое состояние имеет имя и тело, представляющее собой последовательность операторов, описывающих обработку при переходе анализатора в данное состояние, включая:

- объявления локальных переменных;
- операторы присваивания;
- вызовы методов:
    - вызовы функций (например, использование verify для проверки пригодности проанализированных данных);
    - вызовы методов (например, извлечение данных или расчёт контрольных сумм) и других анализаторов (см. параграф 11.10);
- переходы в другие состояния (см. параграф 11.5).

Грамматические правила для операторов parser приведены ниже.

```
parserStatements
        : /* Пусто */
        | parserStatements parserStatement
        ;
parserStatement
        : assignmentOrMethodCallStatement
        | variableDeclaration
        | constantDeclaration
        | parserBlockStatement
        ;
parserBlockStatement
        : optAnnotations '{' parserStatements '}'
        ;
```

Архитектуры могут вносить ограничения на выражения и операторы, которые могут применяться в анализаторах (например, они могут запрещать использование операций типа умножения или ограничивать число используемых локальных переменных).

В терминах ParserModel набор операторов в состоянии выполняется последовательно.

11.5. Операторы переходов

Последним в состоянии анализатора является необязательный оператор transition, который передаёт управления другому состоянию (возможно accept или reject). Синтаксис оператора transition показан ниже.

```
transitionStatement
        : /* Пусто */
        | TRANSITION stateExpression
        ;
stateExpression
        : name ';'
        | selectExpression
        ;
```

Выполнение оператора перехода вызывает оценку выражения stateExpression и передачу управления результирующему состоянию.

В терминах ParserModel семантику оператора transition можно выразить, как

```
        goto eval(stateExpression)
```

Например, оператор

```
        transition accept;
```

прерывает выполнение текущего анализатора и незамедлительный переход в состояние accept.

Если тело блока состояния не заканчивается оператором transition, предполагается оператор

```
        transition reject;
```

11.6. Выражения для выбора

Результатом выражения select является состояние. Синтаксис выражения select показан ниже.

```
selectExpression
        : SELECT '(' expressionList ')' '{' selectCaseList '}'
        ;
selectCaseList
        : /* Пусто */
        | selectCaseList selectCase
        ;
selectCase
        : keysetExpression ':' name ';'
        ;
```

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

В терминах ParserModel смысл выражения select

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

определяется псевдокодом

```
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 внутри оператора select были известны в момент компиляции. Выражения оцениваются по порядку сверху вниз, как показано в приведённом выше псевдокоде. Первый ключ, который содержит значение, соответствующее аргументу select, определяет результирующее состояние. Если совпадений не найдено, выполнение в процессе работы завершается генерацией ошибки со стандартным кодом error.NoMatch.

Отметим, что это предполагает недоступность всех вариантов, расположенных после метки default или _ и компилятору следует выдавать предупреждение в случае обнаружения недоступных меток. Это является важным отличием между выражениями select и операторами switch, используемыми во многих языках программирования, поскольку keyset в выражении select могут «перекрываться».

Типичным примером использования выражения select является сравнение значения недавно извлечённого поля заголовка с набором постоянных значений, как показано ниже.

```
header IPv4_h { ... bit<8> protocol; ... }
struct P { ... IPv4_h ipv4; ... }
P headers;
select (headers.ipv4.protocol) {
        8w6     : parse_tcp;
        8w17    : parse_udp;
```

```
        _       : accept;
}
```

Например, для обнаружения зарезервированных портов TCP (номер < 1024) можно задать выражение

```
select (p.tcp.port) {
        16w0 &&& 16w0xFC00: well_known_port;
        _: other_port;
}
```

Выражение 16w0 &&& 16w0xFC00 описывает набор 16-битовых значений, в которых шесть старших битов имеют значение 0.

11.7. Оператор проверки

Оператор verify обеспечивает простую форму обработки ошибок и может вызываться только внутри parser. Синтаксически оператор используется как показанная ниже внешняя функция.

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

Если первый аргумент имеет значение true, выполнение оператора происходит без побочных эффектов. Однако в случае значения первого аргумента false оператор вызывает незамедлительный переход в состояние reject с прерыванием процесса анализа, а для parserError этого анализатора устанавливается значение второго аргумента.

В терминах ParserModel семантика оператора может быть представлена в виде

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

11.8. Извлечение данных

Библиотека ядра P4 содержит приведённое ниже объявление встроенного внешнего (extern) типа packet_in, который представляет входящие сетевые пакеты. Тип packet_in является специальным и пользователь не может явно создавать экземпляры этого типа. Вместо этого архитектура предоставляет отдельный экземпляр для каждого аргумента packet_in при создании 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();     // Этот метод может быть не доступен в некоторых архитектурах.
        void advance(bit<32> bits);
}
```

Для извлечения данных из пакета, представленного аргументом b типа packet_in, анализатор вызывает методы извлечения данных b. Есть два варианта методов извлечения - вариант с одним аргументом для работы с заголовками фиксированного размера и вариант с двумя аргументами для заголовков переменного размера. Поскольку эти операции могут приводить к отказам при проверках во время работы (см. ниже), эти методы могут вызываться только внутри анализаторов.

При извлечении данных в битовую строку или целое число первый извлечённый из пакета бит помещается в старший бит целого числа.

Некоторые платформы могут обрабатывать пакеты «на лету» (cut-through), т. е. обработка начинается ещё до того, как станет известен размер пакета. На таких платформах вызов метода packet_in.length() не может быть реализован и попытки таких вызовов следует помечать как ошибки (во время компиляции или при попытке загрузки скомпилированной программы P4 в целевую платформу, которая не поддерживает данный метод).

В терминах ParserModel семантику packet_in можно представить с использованием показанной ниже модели абстрактного пакета.

```
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; }
}
```

11.8.1. Извлечение полей фиксированного размера

Метод извлечения с одним аргументом используется для заголовков фиксированного размерами и объявляется в P4, как показано ниже.

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

Выражение headerLeftValue должно оцениваться в l-значение (см. параграф 6.6) типа header с фиксированным размером. При успешном выполнении этого метода headerLvalue заполняется данными из пакета него устанавливается validity = true. Метод может давать отказы по разным причинам (например, при нехватке в пакете битов для заполнения соответствующего заголовка).

Например, приведённый ниже фрагмент кода извлекает заголовок Ethernet.

```
struct Result { ... Ethernet_h ethernet; ... }
parser P(packet_in b, out Result r) {
```

```
        state start {
                b.extract(r.ethernet);
        }
}
```

В терминах ParserModel семантика извлечения с одним аргументом задаётся приведённым ниже псевдокодом с использованием данных из класса пакета, определённого выше. Специальный идентификатор valid$ служит для указания скрытого бита пригодности в заголовке, isNext$ для указания того, что l-значение получено 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;
}
```

11.8.2. Извлечение полей переменного размера

Двухаргументное извлечение предназначено для заголовков переменного размера и декларируется в P4 как показано ниже.

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

Выражение headerLvalue должно быть l-значением, представляющим заголовок с одним полем varbit. Выражение variableFieldSize должно оцениваться в значение bit<32>, которое указывает число битов, извлекаемых в уникальное поле varbit данного заголовка (это значение указывает размер поля varbit, а не всего заголовка).

В терминах ParserModel семантика двухаргументного извлечения может быть выражена приведённым ниже псевдокодом.

```
void packet_in.extract<T>(out T headerLvalue, in bit<32> variableFieldSize) {
        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 однопроходный разбор заголовка IPv4 путём его расщепления на две части.

```
// 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 {
        ...
        IPv4_no_options_h       ipv4;
        IPv4_options_h ipv4options;
}
error { InvalidIPv4Header }
parser Top(packet_in b, out Parsed_headers headers) {
        ...
        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 определения
                // числа извлекаемых битов.
                b.extract(headers.ipv4options,
```

```
                              (bit<32>)(((bit<16>)headers.ipv4.ihl - 5) * 32));
            transition dispatch_on_protocol;
        }
}
```

11.8.3. Взгляд вперёд

Метод lookahead, обеспечиваемый абстракцией пакета packet_in, оценивает набор битов без перемещения указателя nextBitIndex. Подобно извлечению, этот метод будет приводить к переходу в состояние reject и возврату ошибки, если в пакете нет достаточного числа битов. Вызов метода lookahead показан ниже.

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

T должен быть типом с фиксированным размером. При успешном выполнении результат lookahead передаётся в значении типа T.

В терминах ParserModel семантика lookahead представляется 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 8.16 также иллюстрирует использование метода 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;
        }
}
...
state parse_tcp_option_sack {
        bit<32> n = (bit<32>)b.lookahead<Tcp_option_sack_top>().length;
        b.extract(vec.next.sack, n);
        transition start;
}
```

11.8.4. Пропуск битов

P4 обеспечивает два способа пропуска некоторого числа битов входящего пакета без их передачи в заголовок.

Первым способом является передача в переменную _ с явно заданным типом данных.

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

Другой способ используется в тех случаях, когда число пропускаемых битов известно заранее. В терминах ParserModel это можно выразить приведённым ниже псевдокодом.

```
void packet_in.advance(bit<32> bits) {
        lastBitNeeded = this.nextBitIndex + bits;
        ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
        this.nextBitIndex += bits;
}
```

11.9. Стеки заголовков

Стек заголовков имеет два свойства - next и last, которые можно использовать при анализе. Рассмотрим приведённое ниже определение стека для представления заголовков пакета, содержащего 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-значение типа Mpls_h, которое представляет один элемент стека MPLS. Изначально mpls.next указывает на первый элемент стека и автоматически перемещается на следующий элемент при каждом успешном вызове для извлечения данных. Свойство mpls.last указывает на элемент, непосредственно предшествующий элементу next, если тот существует. Попытка доступа к элементу mpls.next, когда значение счётчика nextIndex в стеке не меньше размера этого стека, вызывает переход в состояние reject и установке ошибки error.StackOutOfBounds. Аналогично, попытка доступа к mpls.last при nextIndex = 0 вызывает переход в состояние reject и установку ошибки error.StackOutOfBounds.

Ниже приведён пример упрощённого анализатора для разбора 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; // Это создаёт цикл
                    1: parse_ipv4;
                }
        }
        // Остальные состояния опущены
}
```

11.10. Субанализаторы

P4 позволяет анализаторам пользоваться услугами других анализаторов подобно вызову подпрограмм. Для обращения к другому анализатору сначала требуется создать его экземпляр, а затем обращаться к нему с помощью метода apply.

Ниже приведён пример вызова субанализатора.

```
parser callee(packet_in packet, out IPv4 ipv4) { ...}
parser caller(packet_in packet, out Headers h) {
        callee() subparser;    // Экземпляр вызываемого анализатора
        state subroutine {
                subparser.apply(packet, h.ipv4);      // Вызов субанализатора
        }
}
```

Семантика вызова субанализатора показана ниже.

- Состояние вызывающего анализатора делится на два полусостояния в операторе вызова анализатора.
- Верхнее полусостояние включает переход в состояние start субанализатора.
- Состояние accept субанализатора отождествляется с нижней половиной текущего состояния.
- Состояние reject субанализатора отождествляется с состоянием reject текущего анализатора.

Диаграмма этого процесса представлена на рисунке 9.

Поскольку P4 требует объявления до использования, создание рекурсивных (или взаимно рекурсивных) анализаторов невозможно.

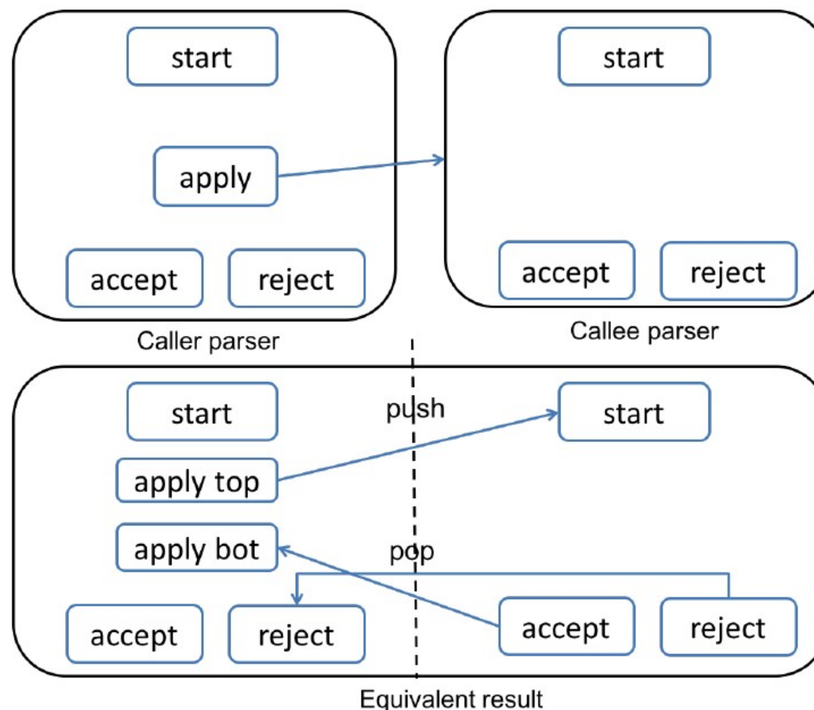

Рисунок 9. Семантика вызова субанализатора - наверху исходная программа, внизу - эквивалент.

Архитектуры могут вносить (статические или динамические) ограничения на число состояний, через которые анализатор может проходить при обработке каждого пакета. Например, компилятор для конкретной платформы может отвергать анализаторы, содержащие петли, которые не могут быть развёрнуты в момент компиляции или содержат циклы, не перемещающие указатель. Если анализатор прерывает выполнение динамически по причине завершения отведённого на разбор времени, анализатору следует перейти в состояние reject и установить стандартную ошибку error.ParserTimeout.

12. Блоки управления

Анализаторы P4 отвечают за извлечение битов из пакетов в заголовки. Этими заголовками (и другими метаданными) можно манипулировать в блоках управления. Тело блока управления похоже на традиционную императивную

программу. Внутри тела могут вызываться элементы «сопоставление-действие» (СД) для выполнения требуемых преобразований. Элементы СД представляются в P4 конструкциями, которые называют таблицами.

Синтаксически блок управления объявляется с именем, параметрами и последовательностью объявлений констант, переменных, действий, таблиц и других экземпляров.

```
controlDeclaration
        : controlTypeDeclaration optConstructorParameters
        /* controlTypeDeclaration не может содержать параметры типа */
        '{' controlLocalDeclarations APPLY controlBody '}'
        ;
controlLocalDeclarations
        : /* Пусто */
        | controlLocalDeclarations controlLocalDeclaration
        ;
controlLocalDeclaration
        : constantDeclaration
        | variableDeclaration
        | actionDeclaration
        | tableDeclaration
        | instantiation
        ;
controlBody
        : blockStatement
        ;
```

Внутри блока управления можно создать экземпляр анализатора. Описание параметров optConstructorParameters, которые могут использоваться для построения параметризованного блока управления, дано в разделе 13.

В отличие от объявления типа элемента управления объявление элемента управления не может быть базовым (generic). Поэтому приведённое ниже объявление будет некорректным.

```
control C<H>(inout H data) { ... }
```

P4 не поддерживает исключительных потоков управления внутри блока управления. Единственным оператором, который оказывает нелокальное воздействие на поток управления, является оператор exit, вызывающий незамедлительное прерывания выполнения включающего его блока управления. Т. е. здесь нет эквивалента оператора verify или состояния reject от анализаторов. Поэтому вся обработка ошибок должны явно выполняться программистом.

В оставшейся части этого раздела описаны основные компоненты блока управления, начиная с действий.

12.1. Действия

Действие (action) представляет собой фрагмент кода, который может считывать и записывать обрабатываемые данные. Действие может включать элементы данных, которые будут записываться уровнем управления и считываться уровнем данных. Действия являются основным средством динамического влияния уровня управления на поведение уровня данных. Абстрактная модель действия представлена на рисунке 10.

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

Синтаксически действие представляет собой функцию, которая не возвращает значения. Действия могут объявляться внутри блока управления и в этом случае такие действия могут применяться только внутри экземпляров этого блока.

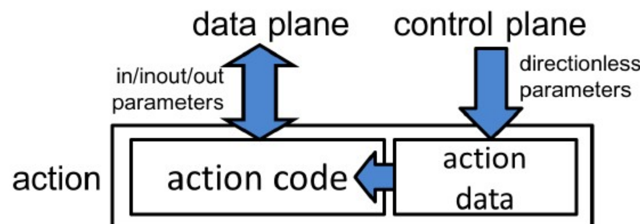

Рисунок 10. Действия содержат код и данные. Код является программой P4, а данные устанавливаются уровнем управления. Привязка параметров выполняется уровнем данных.

Ниже приведён пример объявления действия.

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

Параметры действия могут иметь тип extern. Параметры без направления (например, port в предыдущем примере) указывают «данные действия». Все такие параметры должны указываться в конце списка параметров. При использовании в таблице СД (см. параграф 12.2.1.2) эти параметры будут предоставляться уровнем управления.

Тело действия состоит из последовательности операторов и объявлений. Операторы switch не разрешены внутри действий - грамматика разрешает их, но семантическая проверка должна отвергать. Некоторые платформы могут вносить дополнительные ограничения для тела действий (например, разрешать только линейный код без условных операторов и выражений).

12.1.1. Вызовы операций

Действия могут выполняться двумя способами, описанными ниже.

- Неявно в процессе обработки таблиц СД.

- Явно из блока управления или другого действия. В последнем случае значения для всех параметров действия должны быть представлены явно, включая значения параметров без направления.

12.2. Таблицы

Таблица описывает элемент СД, структура которого показана на рисунке 11. Обработка пакета с использованием таблицы СД включает следующие этапы:

- создание ключа;
- поиск ключа в таблице (сопоставление), результат которого определяет действие;
- выполнение действия (действие) над входными данными, приводящее к их изменениям.

Объявление таблицы создаёт один экземпляр таблицы. Для получения множества экземпляров они должны быть объявлены внутри блока управления, который сам создаётся во множестве экземпляров.

Таблица поиска (look-up table) представляет собой конечное отображение, содержимым которого асинхронно манипулирует (чтение-запись) уровень управления целевой платформы через отдельный интерфейс API уровня управления (см. рисунок 11). Следует подчеркнуть, что термин «таблица» обозначает табличные объекты в программах P4, а также внутренние таблицы поиска в целевых платформах. Для чёткого разделения этих случаев при необходимости будет использоваться термин «элемент СД».

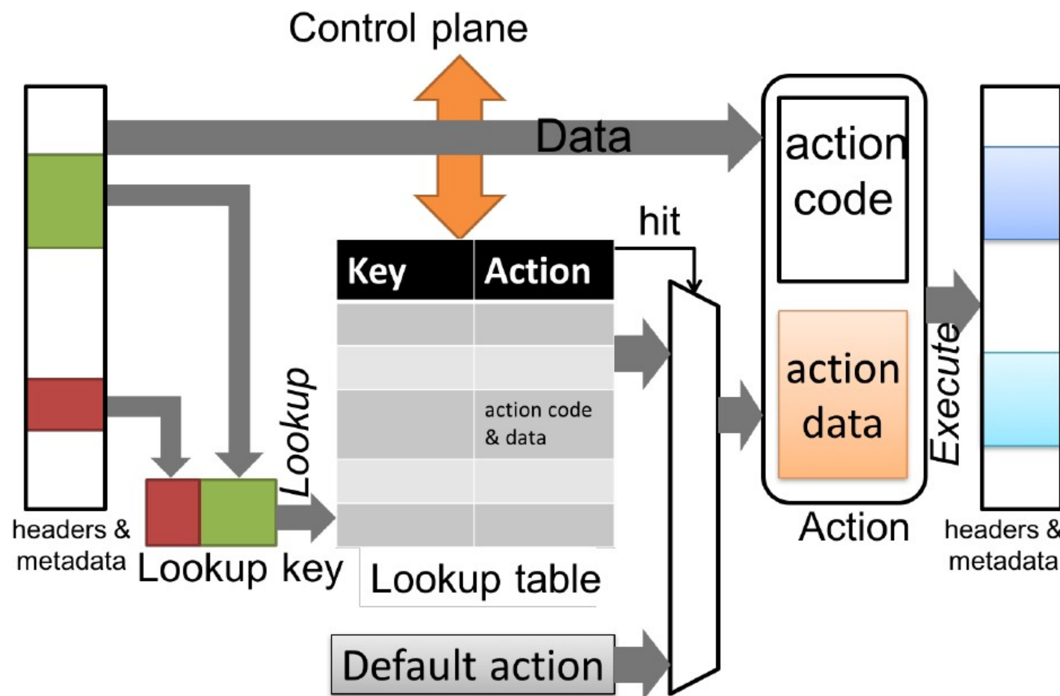

Рисунок 11. Поток данных блока СД.

Синтаксически таблица определяется в терминах набора свойств «ключ-значение». Некоторые из этих свойств являются «стандартными», но набор в целом может быть расширен специфическими для целевой платформы компиляторами.

```
tableDeclaration
        : optAnnotations TABLE name '{' tablePropertyList '}'
        ;
tablePropertyList
        : tableProperty
        | tablePropertyList tableProperty
        ;
tableProperty
        : KEY '=' '{' keyElementList '}'
        | ACTIONS '=' '{' actionList '}'
        | CONST ENTRIES '=' '{' entriesList '}' /* Неизменные записи */
        | optAnnotations CONST IDENTIFIER '=' initializer ';'
        | optAnnotations IDENTIFIER '=' initializer ';'
        ;
```

Стандартные свойства таблицы включают:

- ключ (key) - выражение, определяющее как вычисляется ключ для поиска в таблице;
- действия (actions) - список действий, которые могут быть найдены в таблице;
- необязательное действие по умолчанию (default_action) указывает действие, выполняемое в тех случаях, когда поиск ключа в таблице не дал результата.

Компилятор может устанавливать default_action = NoAction (а также помещать его в список действий) для таблиц, в которых не определено свойство default_action. Это согласуется с семантикой, приведённой в параграфе 12.2.1.3. В данном документе предполагается применение такого преобразования, поэтому все таблицы имеют свойство default_action.

Дополнительно таблица может включать определяемые архитектурой свойства (см. параграф 12.2.1.5). Свойства, помеченные как постоянные (const) не могут динамически изменяться уровнем управления. Свойства key и actions постоянные всегда, поэтому ключевое слово const для них не используется.

12.2.1. Свойства таблиц

12.2.1.1. Ключи

Ключ (key) является свойством таблицы, которое задаёт значения уровня данных, которые следует использовать при поиске записи в таблице. Ключ представляет собой список пар вида (e : m), где выражение e описывает данные для сопоставления с таблицей, а константа m указывает тип сопоставления match_kind, определяющий алгоритм поиска в таблице (см. параграф 7.1.3).

```
keyElementList
        : /* Пусто */
        | keyElementList keyElement
        ;
keyElement
        : expression ':' name optAnnotations ';'
        ;
```

В качестве примера рассмотрим фрагмент программы

```
table Fwd {
        key = {
                ipv4header.dstAddress  : ternary;
                ipv4header.version           : exact;
        }
        ...
}
```

Здесь ключ состоит из двух полей заголовка ipv4header - dstAddress и version. Константа match_kind служит 3 целям:

- задаёт алгоритм, используемый для сопоставления значений уровня данных с записями таблицы;
- используется в API уровня управления для заполнения таблицы во время работы;
- используется компилятором при выделении ресурсов для таблицы.

Библиотека ядра P4 содержит три предопределённых идентификатора match_kind.

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

Эти идентификаторы соответствуют одноимённым типам сопоставления в $P4_{14}$. Семантика этих аннотаций не требуется для описания поведения абстрактной машины P4, а способ их использования влияет лишь на API уровня управления и реализацию таблицы поиска. С точки зрения программы P4 таблица просмотра является абстрактным конечным отображением, которое позволяет по данному ключу определить действие или «пропуск» (miss), как описано в параграфе 12.2.3.

Если таблица не имеет свойства key, она будет не таблицей поиска, а просто принятым по умолчанию действием (т. е., связанная таблица поиска будет просто пустым отображением).

Каждый ключ может иметь необязательную аннотацию @name, которая служит для создания видимого уровню управления имени поля key.

12.2.1.2. Действия

Таблица должна объявлять все возможные действия, которые могут появляться в связанной таблице просмотра или принятом по умолчанию действии. Это выполняется с помощью свойства actions, значением которого всегда является список actionList.

```
actionList
        : actionRef ';'
        | actionList actionRef ';'
        ;
actionRef
        : optAnnotations name
        | optAnnotations name '(' argumentList ')'
        ;
```

Для иллюстрации вернёмся к программе VSS из параграфа 5.3:

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

- Записи в таблице smac могут содержать два разных действия - Drop_action и Rewrite_mac.

- Действие Rewrite_smac имеет один параметр sourceMac, который задаётся уровнем управления.

Каждое действие в списке для таблицы должно иметь своё отличающееся от других имя. Например, приведённый ниже фрагмент программы является некорректным, поскольку в нем используются повторяющиеся имена.

```
action a() {}
control c() {
	action a() {}
	// Недопустимая таблица - два действия имеют одинаковые имена
	table t { actions = { a; .a; } }
}
```

Каждый параметр действия, имеющий направление (in, inout, out) должен быть привязана в спецификации списка действий и наоборот, параметры без направления не могут иметь привязки. Выражения, представленные в качестве аргументов действия, не оцениваются до вызова этого действия.

```
action a(in bit<32> x) { ...}
bit<32> z;
action b(inout bit<32> x, bit<8> data) { ...}
table t {
	actions = {
		// a; - недопустимо, параметр x должен быть привязан
		a(5);  // Привязка в a параметра x к значению 5
		b(z);  // Привязка в b параметра x к значению z
		// b(z, 3); - недопустимая привязка параметр данных без направления
		// b(); - недопустимо, параметр x должен быть привязан
	}
}
```

12.2.1.3. Действие по умолчанию

Принятое по умолчанию действие для таблицы определяет действие, вызываемое автоматически в тех случаях, когда поиск по ключу в таблице не дал результата.

При наличии свойства default_action оно должно указываться после свойства action и может быть объявлено как постоянное (const), которое уровень управления не может динамически изменять. Используемое по умолчанию действие должно быть одним из включённых в список действий. В частности, выражения, передаваемые в качестве параметров in, out или inout, должны быть синтаксически идентичны выражениям, используемым в одном из элементов списка действий.

Например, для приведённой выше таблицы можно установить по умолчанию приведённое ниже действие (помеченное как постоянное).

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

Отметим, что выбранное для использования по умолчанию действие должно предоставлять аргументы для привязки параметров уровня управления (т. е. параметров без направления), поскольку действие синтезируется в момент компиляции. Выражения, представленные в качестве аргументов для параметров с направлением (in, inout, out) оцениваются при вызове действия, тогда как оценка выражений, представленных в качестве аргументов для параметров без направления, выполняется при компиляции.

В продолжение примера из предыдущего параграфа ниже представлен фрагмент, содержащий пригодную и непригодную спецификацию принятого по умолчанию действия для таблицы t.

```
default_action = a(5); // OK - нет параметров уровня управления
// default_action = a(z); - недопустимо, в a параметр x уже привязан к значению 5
default_action = b(z,8w8); // OK - привязка параметра данных в b к значению 8w8
// default_action = b(z); - недопустимо, в b параметр данных не привязан
// default_action = b(x, 3); - недопустимо, в b параметр x привязан к x вместо z
```

Если таблица не содержит свойства default_action и не найдено соответствующей данному пакету записи, таблица не оказывает влияния на пакет и его обработка продолжается в соответствии с императивным потоком управления программы.

12.2.1.4. Записи таблицы

Хотя записи таблиц обычно создаются уровнем управления, таблицы могут также инициализироваться с набором записей в момент компиляции. Это полезно в ситуациях когда таблицы служат для реализации фиксированных записей таблицы, определяемых алгоритмом, статически разрешающих выражения этого алгоритма непосредственно в P4, что позволяет компилятору определить фактическое использование таблицы и принять более эффективные решения по распределению ограниченных ресурсов платформы. Записи, объявленные в исходном коде P4, устанавливаются в таблицах при загрузке программы на платформе.

Синтаксис определения записей таблицы показан ниже.

```
tableProperty
	: const ENTRIES '=' '{' entriesLlist '}' /* Неизменные записи */
entriesList
	: entry
	| entryList entry
entry
	: keysetExpression ':' actionRef optAnnotations ';'
```

Записи таблицы являются неизменными (const), т. е. уровень управления может лишь считывать их, не имея возможности менять или удалять. Это позволяет сохранить неизменными записи таблицы, заданные в исходном коде P4. Это решение оказывает существенное влияние на работу программ P4, поскольку им не требуется отслеживать разные типы (изменяемые или неизменные) записей в таблице. В будущих версиях P4 может появиться возможность включения в одну таблицу изменяемых и неизменных записей путём объявления дополнительных свойств записей без ключевого слова const.

Компонента keysetExpression в записи таблицы представляет собой кортеж (tuple), который должен обеспечивать поле для каждого ключа таблицы (см. параграф 12.2.1). Тип ключа должен соответствовать типу элемента в наборе. Элемент ActionRef должен быть действием, которое присутствует в списке действий таблицы, с привязкой всех аргументов.

Сопоставление записей производится в порядке указания в программе и завершается при первом совпадении.

В зависимости от match_kind ключей выражения установки ключей могут определять одну или множество записей. Компилятор будет синтезировать корректное число записей для установки в таблицу. Ограничения платформы могут сужать возможности создания записей. Например, если число синтезированных записей превышает размер таблицы, реализация компилятора может выдавать сообщение об ошибке в зависимости от возможностей 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);
                }
        }
}
```

В этом примере определён набор из 6 записей, которые приводят к вызову действия a_with_control_params. После загрузки программы эти записи устанавливаются в таблице с сохранением порядка их указания в программе.

12.2.1.5. Дополнительные свойства

Объявление таблицы определяет важные интерфейсы уровней управления и данных - ключи и действия. Однако лучший способ реализации таблицы на деле может зависеть от природы записей, которые будут установлены во время работы (например, таблицы могут быть плотно- или малозаселенными, реализованными как хэш-таблицы, ассоциативная память, дерево и т. п.). Кроме того, некоторые архитектуры могут поддерживать дополнительные свойства таблиц, семантика которых выходит за рамки этой спецификации. Например, архитектуры со статическим распределением ресурсов могут требовать от программиста заранее определять размер таблиц, который будет использоваться при компиляции для выделения требуемых ресурсов хранения. Однако такие зависимые от архитектуры свойства не могут менять семантику поиска в таблицах, который всегда завершается обнаружением (hit) и действием или отсутствием (miss), и могут лишь менять способы интерпретации этих результатов в состояние уровня данных. Такое ограничение нужно для того, чтобы можно было понимать поведение таблиц во время компиляции.

Другой пример свойства реализации может использоваться для передачи дополнительной информации компилятору back-end. Значением этого свойства может быть экземпляр внешнего блока, выбранный из подходящей библиотеки компонент. Например, базовая функциональность конструкций $P4_{14}$ может быть реализована для архитектуры, поддерживающей эту возможность, с помощью конструкции, подобной приведённой ниже.

```
extern ActionProfile {
        ActionProfile(bit<32> size); // Число ожидаемых различных действий
}
table t {
        key = { ...}
        size = 1024;
        implementation = ActionProfile(32);  // Вызов конструктора
}
```

Здесь профиль действия может использоваться для оптимизации в тех случаях, когда таблица имеет большое число записей, но связанные с этими записями действия относятся к небольшому диапазону различающихся значений. Добавление опосредованного уровня позволяет совместно использовать идентичные записи, что может значительно снизить требования к размеру хранилища таблиц.

12.2.2. Вызов элемента СД

К таблице можно обратиться путём вызова метода apply. Вызов этого метода для экземпляра table возвращает значение типа struct с двумя полями. Эта структура синтезируется компилятором автоматически. Для каждой таблицы T компилятор создаёт enum и struct, показанные ниже псевдокодом.

```
enum action_list(T) {
        // Одно поле для каждого действия из списка операций таблицы T
}
struct apply_result(T) {
        bool hit;
        action_list(T) action_run;
}
```

Выполнение метода apply устанавливает в поле hit значение true, если при поиске в таблице было найдено соответствие. Этот флаг может использоваться для ведения потока управления в вызвавшем таблицу блоке управления.

```
if (ipv4_match.apply().hit) {
        // Совпадение
} else {
        // Отсутствие совпадений
}
```

Поле action_run указывает тип выполненного действия (безотносительно к наличию совпадения) и может использоваться в операторе switch.

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

12.2.3. Семантика выполнения блока «сопоставление-действие»

Семантика оператора вызова таблицы приведена ниже.

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

Это можно представить псевдокодом (см. также 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 совпадений в таблице
        result.hit = false;
        RA = m.default_action;       // Использование принятого по умолчанию действия
} 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;
```

12.3. Абстрактная машина конвейера «сопоставление-действие»

Можно описать расчётную модель конвейера СД, встроенного в блок управления - тело блока управления выполняется подобно традиционной императивной программе.

- Во время работы операторы внутри блока выполняются в порядке их размещения в блоке управления.
- Выполнение оператора return ведёт к незамедлительному прерыванию исполнения текущего блока управления и возврату в точку вызова.
- Выполнение оператора exit ведёт к незамедлительному прерыванию исполнения текущего блока управления и всех включающих его блоков управления, которые участвовали в вызове.
- Выполнение таблицы ведёт к исполнению соответствующего блока СД, как описано выше.

12.4. Вызовы элементов управления

P4 позволяет элементам управления обращаться к услугам других элементов управления, подобно вызову подпрограмм. Для обращения к другому блоку управления нужно сначала создать его экземпляр, а затем услуги этого элемента вызываются с помощью его метода apply.

Ниже приведён пример вызова элемента управления.

```
control Callee(inout IPv4 ipv4) { ...}
control Caller(inout Headers h) {
        Callee() instance;                  // Экземпляр вызываемого
        apply {
                instance.apply(h.ipv4);     // Вызов элемента управления
        }
}
```

13. Параметризация

Для поддержки библиотек полезных компонент P4 анализаторы и блоки управления можно дополнительно параметризовать с помощью параметров конструктора.

Рассмотрим ещё раз синтаксис объявления анализатора.

```
parserDeclaration
        : parserTypeDeclaration optConstructorParameters
        '{' parserLocalElements parserStates '}'
        ;
optConstructorParameters
        : /* Пусто */
        | '(' parameterList ')'
        ;
```

Из приведённого фрагмента видно, что объявление анализатора может включать два набора параметров:

- рабочие параметры (parameterList);
- необязательные параметры конструктора (optConstructorParameters).

Параметры конструктора должны указываться без направления (они не могут быть in, out или inout) и при создании экземпляра анализатора должна быть возможность полностью оценить выражения, представленные 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);
                transition select(p.ipv4.protocol) {
                        6: tcp;
                        17: tryudp;
                }
        }
        state tryudp {
                transition select(udpSupport) {
                        false: accept;
                        true : udp;
                }
        }
        state udp {
                ...
        }
}
```

При создании экземпляра GenericParser требуется представить значение параметра udpSupport, как показано ниже.

```
// topParser является GenericParser, где udpSupport = false
GenericParser(false) topParser;
```

13.1. Непосредственный вызов типа

Экземпляры элементов управления и анализаторов зачастую создаются лишь однократно. В целях упрощения объявления анализаторов и элементов управления без параметров конструктора могут применяться непосредственно, как будто это уже экземпляр. Это равносильно созданию и применению локального экземпляра данного типа.

```
control Callee( ... ) { ... }
control Caller( ... )( ... ) {
        apply {
                Callee.apply( ... ); // Callee считается экземпляром
        }
}
```

Определение Caller эквивалентно приведённому ниже

```
control Caller( ... )( ... ) {
        @name("Callee") Callee() Callee_inst; // Локальный экземпляр Callee
        apply {
                Callee_inst.apply( ... );     // Применение Callee_inst
        }
}
```

Эта функция предназначена для упрощения общего случая, когда экземпляр тип создаётся лишь однократно. Для полноты поведение неоднократного прямого вызова одного типа определено, как показано ниже.

- Прямой вызов типа в разных областях действия будет создавать разные локальные экземпляры с различающимися полными именами элементов управления.
- В одной области действия прямой вызов типа будет приводить к разным локальным экземплярам для каждого вызова, однако экземпляры одного типа будут иметь одно глобальное имя за счёт аннотации @name. Если тип содержит контролируемые элементы, непосредственный вызов такого типа более одного раза не допустим, поскольку это будет создавать множество контролируемых элементов с совпадающими полными именами.

Аннотации @name описаны в параграфе 16.3.2.

14. Сборка

Процесс, обратный по отношению к анализу (parsing), называется сборкой (deparsing) или созданием пакета. P4 не включает отдельного языка для сборки пакетов и такая сборка выполняется в блоке управления, имеющем хотя бы один параметр типа packet_out.

Например, приведённый ниже фрагмент кода записывает сначала заголовок Ethernet, а затем заголовок 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 его в конец packet_out только в случае пригодности заголовка. Выдача стека заголовков будет выдавать все элементы стека в порядке возрастания индекса.

14.1. Вставка данных в пакеты

Тип данных packet_out datatype определён в библиотеке ядра P4 и воспроизведён здесь. Он обеспечивает метод добавления данных в конец выходного пакета, называемый emit.

```
extern packet_out {
        void emit<T>(in T data);
}
```

Метод emit поддерживает добавление в конец исходящего пакета данных из заголовка, стека заголовков, структуры или объединения заголовков.

- При использовании метода для заголовка emit добавляет данные из заголовка в конец пакета, если это допустимо. В противном случае не делается ничего (no-op).
- При использовании метода для стека заголовков emit рекурсивно вызывает себя для каждого элемента стека.
- При использовании метода для структуры или объединения заголовков emit рекурсивно вызывает себя для каждого поля.

Метод emit корректно вызывать в выражениях базового типа, а также типов enum или error.

Метод emit можно представить приведённым ниже псевдокодом.

```
packet_out {
        byte[] data;
        unsigned lengthInBits;
        void initializeForWriting() {
                this.data.clear();
                this.lengthInBits = 0;
        }
        /// Добавление данных в конец пакета. Тип T должен быть заголовком, стеком или
        /// объединением заголовков, структурой, рекурсивно созданной из этих 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 варианты для T недопустимы.
        }
}
```

Здесь использован специальный идентификатор valid$ для скрытого бита validity в заголовках и fields$ для указания списка полей структуры или объединения заголовков. Использована также стандартная нотация для обозначения итераций по элементам стека (e : data) и спискам полей для стеков и объединений заголовков (f : data.fields$).

15. Описание архитектуры

Описание архитектуры должно предоставляться производителем платформ в форме исходного кода библиотеки P4, содержащего по крайней мере одно объявление для программы (package), экземпляр которой должен быть создан пользователем при создании программы для этой платформы. В качестве примера можно рассмотреть объявление VSS из параграфа 5.1.

Файл описания архитектуры может определять типы данных, константы, реализации вспомогательных пакетов (helper package) и ошибки. Он должен также объявлять типы программируемых блоков анализа и управления, которые будут использоваться платформой. Программируемые блоки можно группировать в пакеты (package), которые могут быть вложенными.

Поскольку некоторые компоненты платформы могут манипулировать пользовательскими типами, которые ещё не известны в момент объявления платформы, эти типы описываются с использованием переменных типа, которые должны параметрически использоваться в программе (т. е. переменные типа проверяются подобно Java generic, а не шаблонам C++).

15.1. Пример описания архитектуры

Приведённый ниже пример описывает коммутатор с использованием двух программ (package), каждая из которых содержит анализатор, конвейер СД и сборщик (deparser).

```
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<_, IH, OH> map,
                         Deparser<OH> d);
package Egress<T, IH, OH>(Parser<IH> p, Port
                         EPipe<_, IH, OH> map,
                         Deparser<OH> d);
package Switch<T>( // Оператор switch верхнего уровня содержит две программы (package).
        // Типы Ingress.IH и Egress.IH могут быть разными.
        Ingress<T, _, _> ingress,
        Egress<T, _, _> egress
);
```

На основании этих объявлений даже без обращения к подробному описанию платформы программист может получить полезную информацию об архитектуре коммутатора, показанной на рисунке 12.

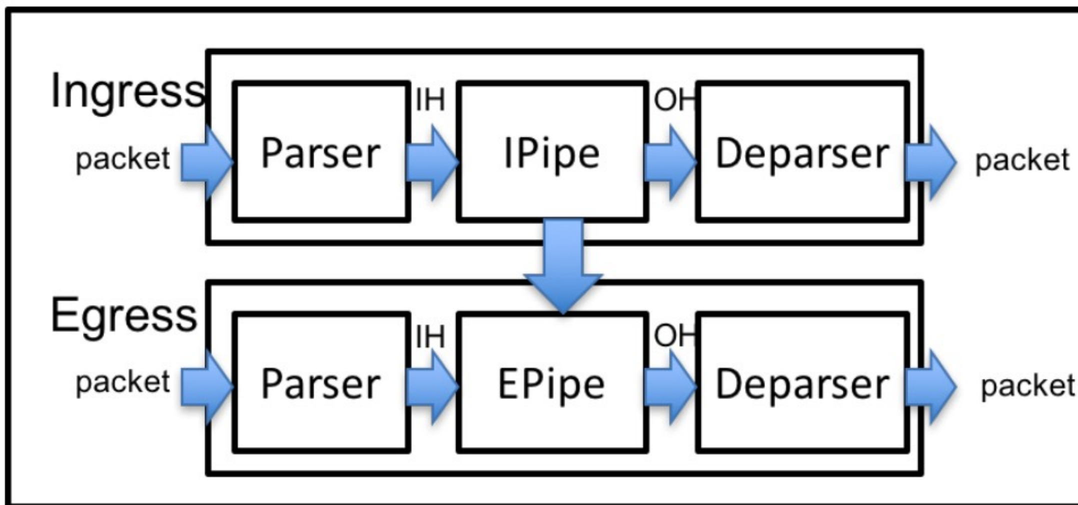

Рисунок 12. Фрагмент архитектуры коммутатора, предполагаемый приведённым выше набором объявлений.

- Коммутатор содержит две отдельных программы (package) - Ingress и Egress.
- Модули Parser, Ipipe и Deparser в пакете Ingress соединены в упорядоченную цепочку. Блок Ingress.IPipe получает данные типа Ingress.IH, которые являются выходом блока Ingress.Parser.
- Аналогично блоки Parser, Epipe и Deparser соединены в цепочку программы Egress.
- Блок Ingress.IPipe соединён с Egress.EPipe, поскольку первый выдаёт значения типа T, служащие входными для второго.
- Отметим, что типы Ingress IH и Egress IH не зависят один от другого. Если мы хотим показать, что они относятся к одному типу, следует вместо объявления IH и OH объявить на уровне коммутатора package Switch<IH, OH, T>.

Отметим, что эта архитектура моделирует коммутатор, который содержит два отдельных канала между входным и выходным конвейером.

- Канал, напрямую передающий данные через аргумент типа T (на программной платформе с общей памятью входного и выходного контроллера это можно реализовать путём прямой передачи указателя; на платформах без общей памяти компилятору предположительно потребуется автоматически создавать код упорядочения).
- Канал опосредованной передачи данных с использованием анализатора и сборщика, которые преобразуют данные в пакет и обратно.

15.2. Пример архитектурной программы

Чтобы создать программу для архитектуры, программа P4 должна создать экземпляр верхнего уровня для package путём передачи значений всех аргументов через переменную main в пространстве имён верхнего уровня. Типы аргументов должны соответствовать типам параметров после подобающей подстановки для переменных типа. Подстановка типов может быть выражена явно с использованием специализации типа или выведена компилятором с использованием алгоритма унификации типа Hindley-Milner.

В качестве примера рассмотрим объявления 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 следующие объявления

```
parser P(packet_in b, out bit<32> index) { ... }
control Pipe1(in bit<32> data) { ... }
control Pipe2(in bit<8> data) { ... }
```

Ниже приведено допустимое объявления для платформы верхнего уровня.

```
Switch(P(), Pipe1()) main;
```

Следующее объявление является некорректным.

```
Switch(P(), Pipe2()) main;
```

Некорректность этого объявления заключается в том, что анализатору P требуется T типа bit<32>, а Pipe2 требует для T тип bit<8>.

Пользователь может также явно указать значения для переменных типа (иначе компьютер будет сам выводить эти значения).

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

15.3. Модель фильтра пакетов

В качестве иллюстрации универсальности языка P4 для описания архитектуры приведём другом пример, в котором моделируется пакетный фильтр, принимающий решения об отбрасывании пакетов на основе результатов анализатора, как показано на рисунке 13.

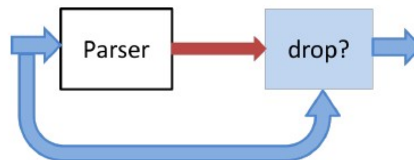

Рисунок 13. Анализатор вычисляет логическое значение для принятия решения об отбрасывании пакета.

Эта модель может использоваться для создания программы фильтрации пакетов в ядре Linux. Например, можно заменить язык tcpdump более мощным языком P4, который способен поддерживать новые протоколы, обеспечивая полную «безопасность типа» (type safety) в процессе обработки пакетов. Для такой платформы компилятор P4 может генерировать программу eBPF[1], которая помещается утилитой tcpdump в ядро Linux и выполняется компилятором EBPF во время работы.

В этом случае целевой платформой является ядро Linux, а архитектурной моделью - фильтр пакетов.

Объявления для этой архитектуры будут иметь вид

```
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);
```

16. Абстрактная машина P4 - оценка

Оценка для программы P4 выполняется в два этапа:

- статическая оценка выполняется во время компиляции путём анализа программы P4 и создания экземпляров всех блоков, имеющих состояния;
- динамическая оценка - во время работы каждый функциональный блок P4 выполняется целиком (неделимо) в изоляции от других при передаче ему управления от архитектуры.

16.1. Известные при компиляции значения

Ниже перечисляются значения, известные в момент компиляции:

- целочисленные, логические и строковые литералы;
- идентификаторы, объявленные в error, enum или match_kind;
- используемый по умолчанию идентификатор;
- поле размера значения с типом header_stack;
- идентификатор _ при использовании в качестве метки в операторе select;
- идентификаторы, которые представляют объявленные типы, действия, таблицы, анализаторы, элементы управления и программы;
- списочные выражения, для которых все компоненты известны при компиляции;
- экземпляры, созданные путём объявления (параграф 9.3) или вызова конструктора;
- выражения (+, -, *, / , %, cast, !, &, |, &&, ||, << , >> , ~ , >, <, ==, !=, <=, >=, ++, [:]), когда все их операнды известны во время компиляции;
- идентификаторы, объявленные как константы с помощью ключевого слова const.

16.2. Оценка во время компиляции

Оценка выполняется в порядке объявления, начиная с пространства имён верхнего уровня.

- Все объявления (например, анализаторы, элементы управления, типы, константы) оценивают себя.

[1]Extended Berkeley Packet Filter - расширенный филь пакетов Беркли.

- Каждая таблица оценивает свои экземпляры.
- Для вызовов конструкторов оцениваются объекты соответствующего типа, сохраняющие состояние. Для этого все аргументы конструкторов оцениваются рекурсивно и привязываются к параметрам конструктора. Аргументы конструктора должны иметь известные при компиляции значения. Порядок оценки аргументов конструктора не должен иметь значения и при любом порядке результаты должны совпадать.
- Оценивается создание экземпляров объектов, сохраняющих состояние.
- При создании экземпляров анализаторов и блоков управления рекурсивно оцениваются все экземпляры в этом блоке, сохраняющие состояние.
- Результатом оценки программы является значение переменной верхнего уровня main.

Отметим, что экземпляры всех значений, сохраняющих состояние, создаются во время компиляции.

В качестве примера рассмотрим приведённый ниже фрагмент программы.

```
// Объявление архитектуры
parser P(...);
control C(...);
control D(...);
package Switch(P prs, C ctrl, D dep);
extern Checksum16 { ...}
// Пользовательский код
Checksum16() ck16; // Экземпляр блока контрольных сумм
parser TopParser(...)(Checksum16 unit) { ...}
control Pipe(...) { ...}
control TopDeparser(...)(Checksum16 unit) { ...}
Switch(TopParser(ck16),
       Pipe(),
       TopDeparser(ck16))main;
```

Оценка этой программы описана ниже.

1. Оцениваются объявления P, C, D, Switch и Checksum16.
2. Экземпляр Checksum16() ck16 оценивается и создаётся объект с именем ck16 и типом Checksum16.
3. Оцениваются объявления TopParser, Pipe и TopDeparser.
4. Оценивается экземпляр переменной main:
   (a) рекурсивно оцениваются аргументы конструктора;
   (b) TopParser(ck16) является вызовом конструктора;
   (c) его аргументы оцениваются рекурсивно и дают в результате объект ck16;
   (d) конструктор оценивает сам себя и это ведёт к созданию экземпляра объекта типа TopParser;
   (e) аналогично оцениваются Pipe() и TopDeparser(ck16) как вызовы конструкторов;
   (f) все аргументы конструктора программы (package) Switch были оценены (это экземпляры TopParser, Pipe и TopDeparser). Их сигнатуры соответствуют объявлению Switch;
   (g) в заключение может быть оценён конструктор Switch и результатом этого будет экземпляр программы Switch (который содержит анализатор TopParser, названный prs в первом параметре Switch, Pipe с именем ctrl и TopDeparser с именем dep).
5. Результатом оценки программы является значение переменной main, которая является экземпляром Switch.

На рисунке 14 показан результат оценки в графической форме. Таким результатом всегда служит граф экземпляров. Имеется один экземпляр Checksum16 с именем ck16, совместно используемый TopParser и TopDeparser. Возможность совместного использования зависит от архитектуры. Компиляторы для конкретных платформ могут требовать использования разных модулей расчёта контрольных сумм для разных блоков.

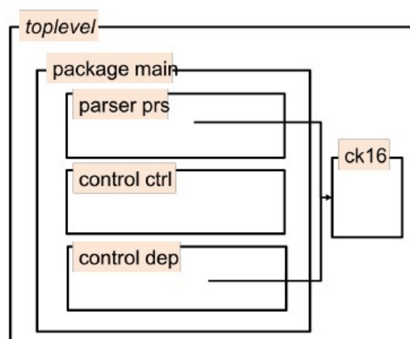

Рисунок 14. Результат оценки.

16.3. Имена для уровня управления

Каждому управляемому элементу, показанному в программе P4, должно быть назначено уникальное полное имя, которое уровень управления может использовать для взаимодействия с этим элементом. Управляемые элементы перечислены ниже:

- таблицы;

- ключи;
- действия;
- экземпляры extern.

Полное имя состоит из локального имени управляемого элемента с префиксом (prepend) в виде полного имени содержащего элемент пространства имён. Поэтому перечисленные ниже программные конструкции, которые включают управляемые элементы, сами должные иметь уникальные полные имена:

- экземпляры элементов управления (control);
- экземпляры анализаторов.

Оценка может создавать множество экземпляров из одного типа и каждый из таких экземпляров должен иметь уникальное полное имя.

16.3.1. Создание имён для управления

Полное имя конструкции создаётся путём конкатенации полных имён включающих конструкций с локальным именем данной конструкции. Полные имена конструкций без охватывающего пространства имён (т. е. определённые в глобальной области действия) совпадают с их локальными именами. Локальные имена управляемых объектов и охватывающих их конструкций выводятся из синтаксиса программы P4, как показано ниже.

16.3.1.1. Таблицы

Для таблиц синтаксическое имя является локальным именем таблицы. Например, таблица

```
control c(...)() {
	table t { ... }
}
```

будет иметь локальное имя t.

16.3.1.2. Ключи

Синтаксически ключи таблицы являются выражениями. Для простых выражений локальное имя может быть создано из самого выражения. В приведённом ниже примере таблица t имеет ключи с именами data.f1 и hdrs[3].f2.

```
table t {
	keys = { data.f1 : exact; }
	hdrs[3].f2 : exact;
}
actions = { ... }
```

В таблице перечислены выражения, для которых локальные имена выводятся из их синтаксических имён.

Вид	*Пример*	*Имя*
Метод isValid()	`h.isValid()`	`"h.isValid()"`
Доступ к массивам	`header_stack[1]`	`"header_stack[1]"`
Константы	`1`	`"1"`
Проекции полей	`data.f1`	`"data.f1"`
Срезы (slice) данных	`f1[3:0]`	`"f1[3:0]"`

Остальные типы выражений должны аннотироваться с @name (параграф 17.1.2) как в приведённом ниже примере.

```
table t {
	keys = { data.f1 + 1 : exact @name("f1_mask"); }
	actions = { ... }
}
```

Здесь аннотация @name("f1_mask") назначает для ключа локальное имя "f1_mask".

16.3.1.3. Действия

Для каждой конструкции action синтаксическое имя является локальным именем. Например,

```
control c(...)() {
	action a(...) { ... }
}
```

создаёт действие с локальным именем a.

16.3.1.4. Экземпляры

Локальные имена для экземпляров extern, parser и control выводятся из способа использования экземпляра. Если экземпляр привязан к имени, это имя становится локальным именем для уровня управления. Например, если элемент управления C объявлен как

```
control C(...)() { ... }
```

и создан его экземпляр

```
C() c_inst;
```

локальным именем экземпляра будет c_inst.

Если же экземпляр создаётся как аргумент, его локальным именем будет имя формального параметра, к которому привязан аргумент. Например, если extern E и control C объявлены как

```
extern E { ... }
control C( ... )(E e_in) { ... }
```

и создан экземпляр

```
C(E()) c_inst;
```

локальным именем экземпляра extern будет e_in.

Если создаваемый экземпляр конструкции передаётся в качестве аргумента программе (package), имя экземпляра выводится из представленного пользователем определения типа, когда это возможно. В приведённом ниже примере локальным именем экземпляра MyC будет c, а локальным именем extern - e2, а не e1.

```
extern E { ... }
control ArchC(E e1);
package Arch(ArchC c);
control MyC(E e2)() { ... }
Arch(MyC()) main;
```

Отметим, что в этом примере архитектура будет представлять экземпляр extern, когда она применяет экземпляр MyC, переданный программе Arch. Полным именем этого экземпляра будет main.c.e2.

Далее рассмотрим пример, показывающий создание имени при наличии множества экземпляров.

```
control Callee() {
        table t { ... }
        apply { t.apply(); }
}
control Caller() {
        Callee() c1;
        Callee() c2;
        apply {
                c1.apply();
                c2.apply();
        }
}
control Simple();
package Top(Simple s);
Top(Caller()) main;
```

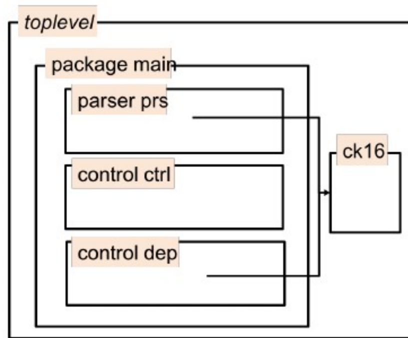

Рисунок 15. Оценка программы, создающей несколько экземпляров одного элемента.

Оценка этой программы во время компиляции создаёт структуру, показанную на рисунке 15. Обратите внимание на наличие двух экземпляров таблицы t. Оба эти экземпляра должны быть видны уровню управления. Для именования объектов в этой иерархии используется путь, образованный именами, содержащими экземпляры. В данном случае таблицы имеют имена s.c1.t и s.c2.t, где s - имя аргумента при создании экземпляра программы, которое выводится из имени соответствующего формального параметра.

16.3.2. Аннотации для имён управляемых элементов

Связанные с уровнем управления аннотации (параграф 17.1.2) могут менять имена, предоставляемые уровню управления, несколькими способами.

- Аннотация @hidden скрывает управляемый элемент от уровня управления. Это единственный случай, когда управляемый элемент не обязан иметь уникальное полное имя.
- Аннотация @name может служить для изменения локального имени управляемого элемента.
- Аннотация @globalname может служить для изменения глобального имени управляемого элемента.

Программы, дающие одно полное имя двум разным управляемым элементам, не допустимы. В частности, следует осторожно использовать аннотации @globalname. Если тип содержит аннотацию @globalname и созданы два его экземпляра, они получат одно полное имя.

16.3.3. Рекомендации

Уровень управления может указывать контролируемый элемент по суффиксу его полного имени, если это обеспечивает уникальность имён в данном контексте. Рассмотрим пример.

```
control c( ... )() {
        action a ( ... ) { ... }
        table t {
                keys = { ... }
                actions = { a; }
        }
}
c() c_inst;
```

Программы уровня управления могут ссылаться на действие c_inst.a при вставке правил в таблицу c_inst.t, поскольку из определения таблицы ясно, о каком действии идёт речь.

Не все однозначные сокращения могут быть рекомендованы. Рассмотрим первый пример из параграфа 16.3. Может возникнуть соблазн указывать s.c1 просто как c1, поскольку в программе нет других экземпляров с именем c1. Однако

это осложнит изменение программы в будущем, поскольку не позволит создавать экземпляры с именем c1 или включать библиотеки P4, содержащие экземпляры с таким именем.

16.4. Динамическая оценка

Динамическая оценка программы P4 организуется архитектурной моделью. Каждая модель архитектуры должна задавать порядок и условия, при которых динамически выполняются различные программные компоненты P4. Например, поток исполнения VSS из параграфа 5.1 имеет вид Parser->Pipe->Deparser.

После вызова блока P4 его исполнение продолжается до прерывания в соответствии с заданной этим документом семантикой.

16.4.1. Модель одновременной обработки

Типичной системе обработки пакетов требуется одновременно выполнять множество логических потоков (thread). имеется по меньшей мере поток уровня управления, который может менять содержимое таблиц. Архитектурным спецификациям следует подробно описывать взаимодействия между уровнями управления и данных. Уровень данных может обмениваться информацией с уровнем управления путём вызова внешних функций и методов. Кроме того, высокопроизводительные системы могут одновременно обрабатывать большое число пакетов (например, в конвейере) или одновременно анализировать один пакет и выполнять операции СД для другого. В этом параграфе описана семантика программ P4 в части одновременного выполнения операций.

Каждый анализатор или блок управления верхнего уровня выполняется в виде отдельного потока (thread). Все параметры блока и все локальные переменные привязаны к этому потоку, т. е. каждый поток имеет свою копию этих ресурсов. Это относится к параметрам packet_in и packet_out анализаторов и сборщиков.

Поскольку блок P4 использует лишь локальное хранилище потока (например, метаданные, заголовки, локальные переменные), его поведение при параллельной работе не отличается от поведения изолированного блока, так как произвольное чередование операторов из разных потоков должно всегда давать неизменный результат.

Внешние блоки, экземпляры которых созданы программой P4Ю являются глобальными и совместно используются всеми потоками. Если блоки extern служат для доступа к состоянию (например, счётчики, регистры) и методы внешнего блока могут считывать и записывать состояние, операции с учётом состояния становятся состязательными. P4 в таких случаях диктует определённое поведение.

- Выполнение операции является неделимым (atomic), т. е. другие потоки могут «видеть» состояние, которое было до начала выполнения операции или после её завершения.
- Вызов метода для экземпляра extern является неделимым.

Чтобы пользователь мог задать неделимое выполнение более крупного блока кода, P4 поддерживает аннотацию @atomic, которая может быть применена к блоку операторов, состояниям анализатора, блокам управления или анализатору в целом.

Рассмотрим пример

```
extern Register { ... }
control Ingress() {
        Register() r;
        table flowlet { /* Чтение состояния r в действии */ }
        table new_flowlet { /* Запись состояния r в действии */ }
        apply {
                @atomic { flowlet.apply(); }
}}
if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TIMEOUT)
        new_flowlet.apply();
```

Эта программа обращается к внешнему объекту r типа Register в операциях, вызываемых из таблиц flowlet (чтение) и new_flowlet (запись). Без аннотации @atomic эти две операции не будут неделимыми и для второго пакета состояние r может быть прочитано до того, как первый сможет его изменить.

Компилятор (backend) должен отклонить программу с блоками @atomic, если он не поддерживает неделимого выполнения последовательности инструкций. Для таких случаев компилятору следует поддерживать диагностику.

17. Аннотации

Аннотации похожи на атрибуты C# и аннотации Java и обеспечивают простой механизм некоторого расширения языка P4 без изменения его грамматики. В каком-то смысле это походе на C #pragma. Аннотации могут добавляться к типам, полям, переменным и т. д. с использованием синтаксиса @ (как показано явно в грамматике P4).

```
optAnnotations
        : /* Пусто */
        | annotations
        ;
annotations
        : annotation
        | annotations annotation
        ;
annotation
        : '@' name
        | '@' name '(' expressionList ')'
        ;
```

17.1. Предопределённые аннотации

Аннотации, имена которых начинаются со строчных букв, зарезервированы для стандартной библиотеки и архитектуры, «стандартные» аннотации определены в этом документе и их список может расширяться. Для

фирменных архитектур предлагается определять аннотации, начинающиеся с префикса производителя (например, организация X может использовать аннотации вида @X_annotation).

17.1.1. Аннотации списка действий таблицы
Приведённые ниже аннотации могут служить для предоставления дополнительной информации о действиях таблицы компилятору и уровню управления. Эти аннотации не имеют аргументов.

- @tableonly - действия с такой аннотацией могут быть лишь в таблице и не могут применяться по умолчанию.
- @defaultonly - действия с такой аннотацией могут лишь применяться по умолчанию и не могут быть в таблице.

```
table t {
        actions = {
                a,                      // Может присутствовать где угодно
                @tableonly b,           // Может присутствовать только в таблице
                @defaultonly c,         // Может присутствовать в принятом по умолчанию действии
        }
        ...
}
```

17.1.2. Аннотации API уровня управления
Аннотация @name говорит компилятору, что нужно использовать другое локальное имя при генерации внешних API, используемых для манипуляций с элементами языка с уровня управления. Аннотация может иметь аргумент в виде строкового литерала. В приведённом ниже примере полное имя таблицы будет c_inst.t1.

```
control c( ... )() {
        @name("t1") table t { ... }
        apply { ... }
}
c() c_inst;
```

Аннотация @globalname похожа на @name, но меняет полное имя (а не только локальное) элемента. В приведённом примере полным именем таблицы будет foo.bar.

```
control c( ... )() {
        @globalname("foo.bar") table t { ... }
        apply { ... }
}
c() c_inst;
```

Аннотация @hidden скрывает управляемый элемент (table, key, action, extern) от уровня управления и не имеет аргументов. Операция эффективно скрывает полное имя элемента (параграф 16.3).

17.1.2.1. Ограничения
Каждый элемент может использовать не более одной аннотации @name, @globalname или @hidden и каждое имя на уровне управления должно указывать не более одного управляемого элемента. Для аннотации @globalname имеется особый случай — если она используется дважды, одно глобальное имя будет указывать на два управляемых объекта.

```
control noargs();
package top(noargs c1, noargs c2);
control c() {
        @globalname("foo.bar") table t { ... }
        apply { ... }
}
top(c(), c()) main;
```

Без аннотации @globalname программа будет создавать два управляемых элемента с полными именами main.c1.t и main.c2.t. Однако аннотация @globalname("foo.bar") меняет имя таблиц t в обоих экземплярах на foo.bar и имена управляемых объектов становятся непригодными.

17.1.3. Аннотации управления параллельной работой
Аннотация @atomic, описанная в параграфе 16.4.1, может служить для задания неделимого исполнения блока кода.

17.2. Специфические для платформы аннотации
Каждая реализация компилятора P4 может определять дополнительные аннотации, относящиеся к целевой платформе компилятора. Синтаксис таких аннотаций должен соответствовать приведённому выше описанию. Семантика этих аннотаций будет зависеть от платформы. Аннотации могут применяться как pragma в других языках.

Компиляторам P4 следует поддерживать перечисленные ниже уведомления.

- Ошибки при некорректном использовании аннотаций (например, для аннотации задан параметр, но она используется без него или с недопустимым типом параметра).
- Предупреждения при неизвестных аннотациях.

Приложение A. Зарезервированные слова P4
В таблице перечислены все зарезервированные слова P4. Некоторые идентификаторы считаются ключевыми словами 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	struct	switch
table	transition	true	tuple
typedef	varbit	verify	void

Приложение B. Библиотека ядра P4

Библиотека ядра P4 содержит объявления, которые могут использоваться большинством программ.

Например, библиотека ядра включает объявления предопределённых внешних объектов packet_in и packet_out, используемых в анализаторах и сборщиках для доступа к данным пакета.

```
/// 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 в стеке заголовков.
   HeaderTooShort,  /// Извлечение слишком большого числа битов в поле varbit.
   ParserTimeout        /// Истекло время работы анализатора.
}
extern packet_in {
   /// Читает заголовок из пакета в поле типа header @hdr с фиксированным размером
   /// и перемещает указатель.
   /// Может вызывать ошибку PacketTooShort или StackOutOfBounds.
   /// @T должно иметь тип header фиксированного размера.
   void extract<T>(out T hdr);
   /// Читает биты из пакета в поле типа header @variableSizeHeader с переменным
   /// размером и перемещает указатель.
   /// @T должно быть заголовком, содержащим в точности 1 поле varbit.
   /// Может вызывать ошибку PacketTooShort, StackOutOfBounds или HeaderTooShort.
   void extract<T>(out T variableSizeHeader, in bit<32> variableFieldSizeInBits);
   /// Читает биты из пакета без перемещения указателя.
   /// @returns - прочитанные из пакета биты.
   /// T может иметь произвольный тип с фиксированным размером.
   T lookahead<T>();
   /// Перемещает указатель на заданной число битов.
   void advance(in bit<32> sizeInBits);
   /// @return - размер пакета в байтах. Этот метод доступен не в каждой архитектуре.
   bit<32> length();
}
extern packet_out {
   /// Запись @data в выходной пакет, пропуск некорректных заголовков и сдвиг указателя.
   /// @T может иметь тип header, header_stack, а header_union или struct с этими типами.
   void emit<T>(in T data);
}
action NoAction() {}
   /// Стандартный типы сопоставления для полей таблицы.
   /// Некоторые архитектуры могут не поддерживать эти сопоставления.
   /// Архитектура может объявлять свои сопоставления.
match_kind {
   /// Точное совпадение битов.
   exact,
   /// Тройное сопоставление с использованием маски.
   ternary,
   /// Самый длинный из совпадающих префиксов.
   lpm
}
```

Приложение C. Контрольные суммы

$P4_{16}$ не имеет встроенных конструкций для работы с контрольными суммами пакетов. Предполагается, что операции с контрольными суммами могут быть реализованы в виде внешних объектов библиотек конкретных платформ. Стандартной библиотеке архитектуры следует включать модули контрольных сумм.

Например, можно предоставить модуль инкрементного расчёта контрольных сумм Checksum16 (см. описание VSS в параграфе 5.2.4) для расчёта 16-битовый дополнений до 1 с использованием внешнего объекта с сигнатурой типа

```
extern Checksum16 {
       Checksum16();                 // Конструктор
       void clear();                 // Подготовка модуля к расчёту.
       void update<T>(in T data);    // Добавление данных в контрольную сумму.
       void remove<T>(in T data);    // Исключение данных их контрольной суммы.
       bit<16> get();          // Получение контрольной суммы для данных, добавленных после
                         // последнего сброса
}
```

Проверка контрольной суммы IP может быть выполнена в анализаторе, как показано ниже.

```
ck16.clear();                 // Подготовка блока контрольных сумм
ck16.update(header.ipv4); // Запись заголовка
verify(ck16.get() == 16w0, error.IPv4ChecksumError); // Проверка контрольной суммы 0
```

Генерация контрольной суммы IP может быть выполнена, как показано ниже.

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

Кроме того, некоторые архитектуры коммутаторов не выполняют проверку контрольных сумм и лишь делают инкрементное обновление с учётом изменения пакета. Это можно сделать с помощью приведённого фрагмента P4.

```
ck16.clear();
ck16.update(header.ipv4.hdrChecksum);          // Исходная контрольная сумма
ck16.remove( { header.ipv4.ttl, header.ipv4.proto } );
header.ipv4.ttl = header.ipv4.ttl - 1;
ck16.update( { header.ipv4.ttl, header.ipv4.proto } );
header.ipv4.hdrChecksum = ck16.get();
```

Приложение D. Нерешенные вопросы

Есть множество открытых вопросов, которые обсуждаются в рабочей группе P4. Краткое описание этих вопросов приведено здесь. Мы ждём информации об этих вопросах от сообщества и призываем к экспериментам с различными реализациями компиляторов для включения результатов в будущие спецификации.

D.1. Переносимая архитектура коммутатора

Переносимость и композитность имеют важное значение для долгосрочного успеха P4. Композитность означает возможность реализации различных функций типа сетевой телеметрии по основному каналу (INT[1]), виртуализации сетей, и балансировки нагрузки в разных программах P4, написанных для PSA[2], с взаимодействием, управляемым из программ верхнего уровня. Переносимость означает работу реализации некой функции (типа INT) в P4 на разных архитектурах, поддерживающих PSA. С этой целью рабочая группа по архитектуре разрабатывает спецификацию, которая позволит создавать программы P4 для работы на разных платформах.

D.2. Обобщение оператора switch

$P4_{16}$ включает операторы switch (параграф 10.7) и выражения select (параграф 11.6). В текущей версии они реально различаются и операторы должны оцениваться в значение состояния.

Предлагаются обобщённые операторы switch для соответствия распространенным языкам программирования — многокомпонентные условия в которых выполняется первый из совпадающих вариантов.

```
switch(e1,...,en) {
        pat_1 : stmt1;
        ...
        pat_m : stmtm;
}
```

Здесь проверяемое значение задано кортежем (e1,...,en), а варианты указаны выражениями-шаблонами, которые обозначают наборы значений. Значение соответствует варианту, если оно попадает заданный шаблоном набор. В отличие от C и C++, здесь нет оператора break и просмотр вариантов не завершается при обнаружении соответствия, если только нет специального оператора «выхода», связанного с данным вариантом.

Это предназначено для фиксации стандартной семантики операторов switch, а также общей идиомы в анализаторах P4, где эти операторы применяются для управления переходами между различными состояниями анализатора в зависимости от значения одного или множества уже проанализированных полей. Используя операторы switch, можно также обобщить устройство анализаторов, избавляясь от выбора и ослабляя большинство ограничений для типов операторов, разрешённых в состоянии. В частности, разрешено использование условных операторов и select с произвольным уровнем вложенности. Данный язык можно транслировать в более ограниченные версии, где тело каждого состояния образовано последовательностью объявлений переменных, присваиваний значений и вызовов методов, завершающейся одним оператором transition для перехода в новое состояние.

Мы также упрощаем обработку hit/miss в таблицах и действий в блоках управления за счёт генерации неявных типов для действий и результатов.

Контраргументом для этого предложения может служить то, что семантика select в анализаторе существенно отличается от семантики switch, а кроме того эти конструкции уже знакомы программистам и обычно уже эффективно используются для разных платформ.

D.3. Неопределённое поведение

Возможность неопределённого поведения вызывает множество проблем в языках типа C и HTML, включая ошибки и серьёзные уязвимости защиты. Есть несколько случаев когда оценка программы P4 может приводить к неопределённому поведению - параметры out, неинициализированные переменные, обращение к полям непригодных заголовков, выход индекса стека заголовков за допустимые пределы. Мы считаем, что нужно делать все необходимое для предотвращения неопределённого поведения в $P4_{16}$ поэтому предлагается усиливать формулировки спецификации, чтобы по умолчанию программы с неопределённым. поведением были исключены. Принимая во внимание вопросы производительности, предлагается определить флаги компилятора и/или операторы pragma, которые могут отменять безопасное поведение. Тем не менее, мы надеемся, что программисты будут работать над созданием безопасных программ и призываем их к этому.

D.4. Структурированные итерации

Добавление конструкции типа foreach для работы со стеками заголовков смягчает необходимость использования директив препроцессора C для задания размера стека заголовков. Например,

```
foreach hdr in hdrs {
        ... операции над HDR ...
}
```

Поскольку стеки всегда известны статически (в момент компиляции), компилятор может преобразовать оператор foreach в реплицированный код с явными индексами. Это позволяет создавать код независимо от параметризованного размера стека заголовков.

[1]In-band Network Telemetry.
[2]Portable Switch Architecture — переносимая архитектура коммутатора.

Поскольку компилятор может статически определить число операций, которые будут возникать из оператора foreach, он может также отвергнуть программу, если результат будет требовать недоступного объёма ресурсов для действия или разделить код действия с учётом доступных ресурсов.

Приложение E. Грамматика P4

В этом приложении грамматика $P4_{16}$ описана с использованием языка YACC/bison. Грамматика не задаёт приоритет для операций.

Эта грамматика не однозначна, поэтому лексический (lexer) и синтаксический (parser) анализаторы должны взаимодействовать при анализе. В частности, лексический анализатор должен различать два типа идентификаторов:

- ранее введённые имена типов (маркеры TYPE);
- обычные идентификаторы (маркер IDENTIFIER).

Синтаксический анализатор должен использовать таблицу символов для указания лексическому анализатору способа разбора идентификаторов. Например, для приведённого ниже фрагмента программы

```
typedef bit<4> t;
struct s { ...}
t x;
parser p(bit<8> b) { ... }
```

Лексический анализатор должен возвращать следующие типы терминалов:

```
t - TYPE
s - TYPE
x - IDENTIFIER
p - TYPE
b - IDENTIFIER
```

На эту грамматику сильно повлияли ограничений инструмента для генерации синтаксических анализаторов Bison.

Ниже перечислены других постоянных терминалов, используемых в правилах грамматики.

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

Маркер STRING_LITERAL соответствует строковому литералу в двойных кавычках, как описано в параграфе 6.3.3.3.

Все остальные терминалы являются написанием соответствующих ключевых слов заглавными буквами. Например, RETURN будет терминалом, возвращаемым лексическим анализатором при разборе ключевого слова return.

```
p4program
    : /* Пусто */
    | p4program declaration
    | p4program ';'   /* Пустое объявление */
    ;

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

nonTypeName
    : IDENTIFIER
    | APPLY
    | KEY
    | ACTIONS
    | STATE
    ;

name
    : nonTypeName
    | TYPE
    | ERROR
    ;

optAnnotations
    : /* Пусто */
    | annotations
    ;
```

```
annotations
    : annotation
    | annotations annotation
    ;

annotation
    : '@' name
    | '@' name '(' expressionList ')'
    ;

parameterList
    : /* Пусто */
    | nonEmptyParameterList
    ;

nonEmptyParameterList
    : parameter
    | nonEmptyParameterList ',' parameter
    ;

parameter
    : optAnnotations direction typeRef name
    ;

direction
    : IN
    | OUT
    | INOUT
    | /* Пусто */
    ;

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

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

optConstructorParameters
    : /* Пусто */
    | '(' parameterList ')'
    ;

dotPrefix
    : '.'
    ;

/**************************** Анализатор ******************************/
parserDeclaration
    : parserTypeDeclaration optConstructorParameters '{' parserLocalElements parserStates '}'
      /* Нет параметров типа, разрешённых в parserTypeDeclaration */
    ;

parserLocalElements
    : /* Пусто */
    | parserLocalElements parserLocalElement
    ;

parserLocalElement
    : constantDeclaration
    | variableDeclaration
    | instantiation
    ;

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
    ;

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 ')'
    ;

simpleExpressionList
    : simpleKeysetExpression
    | simpleExpressionList ',' simpleKeysetExpression
    ;

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

/*************************** Элементы управления ************************/
controlDeclaration
    : controlTypeDeclaration optConstructorParameters
      /* нет параметров типа, разрешённых в controlTypeDeclaration */
      '{' controlLocalDeclarations APPLY controlBody '}'
    ;

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

controlLocalDeclarations
    : /* Пусто */
    | controlLocalDeclarations controlLocalDeclaration
    ;

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

controlBody
    : blockStatement
    ;

/*************************** Внешние объекты ************************/
externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters '{' methodPrototypes '}'
```

```
    | optAnnotations EXTERN functionPrototype ';'
    ;

methodPrototypes
    : /* Пусто */
    | methodPrototypes methodPrototype
    ;

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

methodPrototype
    : functionPrototype ';'
    | TYPE '(' parameterList ')' ';'
    ;

/************************** Типы ****************************/
typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    ;

prefixedType
    : TYPE
    | dotPrefix TYPE
    ;

typeName
    : prefixedType
    ;

tupleType
    : TUPLE '<' typeArgumentList '>'
    ;

headerStackType
    : typeName '[' expression ']'
    ;

specializedType
    : prefixedTYpe '<' typeArgumentList '>'
    ;

baseType
    : BOOL
    | ERROR
    | BIT
    | BIT '<' INTEGER '>'
    | INT '<' INTEGER '>'
    | VARBIT '<' INTEGER '>'
    ;

typeOrVoid
    : typeRef
    | VOID
    | nonTypeName     // Может быть переменной типа
    ;

optTypeParameters
    : /* Пусто */
    | '<' typeParameterList '>'
    ;

typeParameterList
    : nonTypeName
    | typeParameterList ',' nonTypeName
    ;

typeArg
    : DONTCARE
    | typeRef
    ;

typeArgumentList
    : typeArg
    | typeArgumentList ',' typeArg
    ;

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

```
    | packageTypeDeclaration ';'
    ;

derivedTypeDeclaration
    : headerTypeDeclaration
    | headerUnionDeclaration
    | structTypeDeclaration
    | enumDeclaration
    ;

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

headerUnionDeclaration
    : optAnnotations HEADER_UNION name { structure.declareType(*$3); }
      '{' structFieldList '}'
       { $$ = new IR::Type_Union(@3, *$3, $1, *$6); }
    ;

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

structFieldList
    : /* Пусто */
    | structFieldList structField
    ;

structField
    : optAnnotations typeRef name ';'
    ;

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

errorDeclaration
    : ERROR '{' identifierList '}'
    ;

matchKindDeclaration
    : MATCH_KIND '{' identifierList '}'
    ;

identifierList
    : name
    | identifierList ',' name
    ;

typedefDeclaration
    : annotations TYPEDEF typeRef name ';'
    | TYPEDEF typeRef name ';'
    | annotations TYPEDEF derivedTypeDeclaration name ';'
    | TYPEDEF derivedTypeDeclarationame ';'
    ;

/**************************** Операторы *************************/
assignmentOrMethodCallStatement
    : lvalue '(' argumentList ')' ';'
    | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
    | lvalue '=' expression ';'
    ;

emptyStatement
    : ';'
    ;

returnStatement
    : RETURN ';'
    ;

exitStatement
    : EXIT ';'
    ;

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

// Для поддержки прямого вызова элемента управления или анализатора без создания экземпляра
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
    ;

statementOrDeclaration
    : variableDeclaration
    | constantDeclaration
    | statement
    | instantiation
    ;

/************ Таблицы *************/
tableDeclaration
    : optAnnotations TABLE name '{' tablePropertyList '}'
    ;

tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

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

keyElementList
    : /* Пусто */
    | keyElementList keyElement
    ;

keyElement
    : expression ':' name optAnnotations ';'
    ;

actionList
    : actionRef ';'
    | actionList actionRef ';'
    ;

entriesList
    : entry
    | entryList entry

entry
    : optAnnotations keysetExpression ':' actionRef ';'

actionRef
    : optAnnotations name
```

```
    | optAnnotations name '(' argumentList ')'
    ;

/************************* Действие ********************************/
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 ****************************/
argumentList
    : /* Пусто */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList ',' argument
    ;

argument
    : expression
    ;

expressionList
    : /* Пусто */
    | expression
    | expressionList ',' expression
    ;

member
    : name
    ;

prefixedNonTypeName
    : nonTypeName
    | dotPrefix nonTypeName
    ;

lvalue
    : prefixedNonTypeName
    | lvalue '.' member
    | lvalue '[' expression ']'
    | lvalue '[' expression ':' expression ']'
    ;

%left ','
%nonassoc '?'
%nonassoc ':'
%left OR
%left AND
%left '|'
%left '^'
%left '&'
%left EQ NE
%left '<' '>' LE GE
%left SHL
%left PP '+' '-'
%left '*' '/' '%'
%right PREFIX
%nonassoc ']' '(' '['
%left '.'

// Требуется задать дополнительные предпочтения
expression
    : INTEGER
    | TRUE
    | FALSE
```

```
    | STRING_LITERAL
    | nonTypeName
    | '.' nonTypeName
    | expression '[' expression ']'
    | expression '[' expression ':' expression ']'
    | '{' expressionList '}'
    | '(' expression ')'
    | '!' expression
    | '~' expression
    | '-' expression
    | '+' expression
    | typeName '.' member
    | ERROR '.' member
    | expression '.' member
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '+' expression
    | expression '-' expression
    | expression SHL expression              // <<
    | expression '>''>' expression           // Проверка того, что символы >> смежны
    | expression LE expression        // <=
    | expression GE expression        // >=
    | expression '<' expression
    | expression '>' expression
    | expression NE expression        // !=
    | expression EQ expression        // ==
    | expression '&' expression
    | expression '^' expression
    | expression '|' expression
    | expression PP expression        // ++
    | expression AND expression              // &&
    | expression OR expression        // ||
    | expression '?' expression ':' expression
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
    | typeRef '(' argumentList ')'
    | '(' typeRef ')' expression
    ;
```

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru