NPL - Network Programming Language Specification v1.3

June 11, 2019

Перевод спецификации языка NPL, версия 1.3

Оглавление

Список таблиц

Список рисунков

1. Сфера применения

Этот документ описывает конструкции и применение языка сетевого программирования NPL[1]. Основной целью NPL является описание поведения обработки пакетов на уровне данных (Data Plane Packet Processing) с использованием подходящего набора конструкций. Приложение обработки пакетов в NPL включает конструкции высокого уровня для таких задач, как синтаксический анализ, таблицы «сопоставление-действие», редактирование пакетов. Язык также включает другие конструкции, такие как функции.

Поскольку основным требованием к языку является его отображение на гибко настраиваемое сетевое оборудование, полный набор конструкций сосредоточен на возможностях оборудования. Эти конструкции должны применяться при разработке программ.

Документ предназначен для системных архитекторов, инженеров-проектировщиков и инженеров-программистов, которым нужно понимать логику NPL, вносить изменения в программы NPL или разрабатывать приложения для обработки пакетов. Инженерам-тестировщикам также следует понимать NPL для выполнения эффективных тестов.

Документ не рассматривает архитектуру программируемых устройств и работу компиляторов NPL.

2. Термины

Ниже приведены определения терминов, концепций, символов и сокращений, используемых в документе.

NPL Compiler - компилятор NPL

Состоит компиляторов Front End (FE) и Back End (BE).

Front End (FE) Compiler - компилятор FE

Компонент компилятора, выполняющий синтаксический анализ и проверку корректности исходного кода NPL, а также генерирующий промежуточное представление.

Back End (BE) Compiler - компилятор BE

Компонент компилятора, создающий аппаратный код на основе промежуточного представления (IR).

IR files - файлы IR.

Файлы промежуточного представления.

Constructs - конструкции.

Встроенные компоненты для выполнения определённых функций.

Metadata - метаданные.

Поля шины (bus), заголовка (header) и таблицы (table), которые не создаются в NPL, но присутствуют и доступны.

3. Обзор

Рост программно-определяемых сетей (SDN[2]) повысил уровень ожиданий в части программируемости и автоматизации сетей. Исходно задачей SDN было решение проблем уровня управления в стремлении преодолеть ограничения традиционных моделей управления. Впоследствии пользователям потребовались более гибкие решения, способные адаптироваться к изменению сетевых потребностей, например, поддержка новых наложенных протоколов и расширение возможностей телеметрии. Это расширило сферу применения SDN путём включения задач программирования уровня данных. Однако новые гибкие решения для коммутации должны обеспечивать производительность на полной скорости линии с оптимизацией ресурсов коммутатора и потребляемой мощности.

Хотя для программирования уровня данных можно разработать разные языке, важно обеспечить в них поддержку конструкций, способных программировать современные устройства. Для этого было разработан новый язык программирования - NPL, являющийся открытым языком высокого уровня для эффективного программирования уровня пересылки пакетов. NPL включает конструкции для описания поведения сети, использующие преимущества базового программируемого оборудования.

В своём первом воплощении NPL обеспечивает все требуемые функции для реализации надёжных коммутационных решений. NPL включает широкий набор компонент от типов данных для задания отдельных сигналов управления до конструкций высокого уровня, позволяющих взаимодействовать со сложными аппаратными блоками.

В типичном коммутаторе с фиксированными функциями используется набор таблиц и объектов для обработки пакетов, который сложно изменить после производства коммутатора. В программируемом коммутаторе элементы обработки пакетов может определять пользователь. NPL позволяет задать детали таблиц и других объектов для достижения нужного поведения коммутатора. NPL является специализированным языком программирования уровня данных, основанным на традиционных языках программирования, используемых уровнем управления для настройки путей коммутации данных.

В NPL применяется ряд описанных ниже абстракций.

Data Types - типы данных
Определяет тип поля данных.
Parser - синтаксический анализатор
Идентифицирует разрешённые заголовки в принятых пакетах и извлекает поля таких заголовков.
Logical Bus - логическая шина
Задаёт поля и наложения (overlay) логической шины, соединяющей объекты NPL.
Logical Table (Match Action table) - логическая таблица (сопоставление-действие)

[1] *Network Programming Language.*
[2] *Software Defined Networks.*

Описывает конкретную таблицу с ключами поиска и действиями. NPL поддерживает таблицы index, hash, tcam, lpm, alpm.

Editor - редактор

Обеспечивает возможность добавлять, удалять или заменять заголовки.

Special Function - специальная функция

Механизм для вызова определённой аппаратной функции, которая может считаться интеллектуальной собственностью. Это обеспечивает структурированный механизм взаимодействия с такими функциями без раскрытия их содержимого.

Function - функция

Обеспечивает программируемую логику принятия решений без использования таблицы. Например, функция может служить для выбора среди нескольких результатов «сопоставление-действие» или ключа для сопоставления.

Strength Resolution – выбор среди таблиц

Механизм выбора одной из нескольких таблиц, одновременно (параллельно) обновляющих один объект.

Packet Drop, Packet Trace, Packet Count

Встроенные функции для отбрасывания, трассировки и учёта пакетов.

Create Checksum, Update Packet Length

Встроенные функции для расчёта контрольной суммы и обновления размера пакетов.

Metadata for MA and Parser - метаданные для таблиц и анализаторов

Данные, не создаваемые в NPL, но существующие в процессе работы с пакетом и доступные для использования в NPL.

Язык NPL не привязан к какой-либо аппаратной архитектуре и предназначен для реализации на разных аппаратных платформах, включая ASIC, программируемые NIC[1], FPGA и программные коммутаторы. Хотя определённые конструкции языка предназначены для использования конкретных свойств оборудования, это не препятствует отображению программ на цели, не поддерживающие таких свойств.

Подобно другим языкам высокого уровня, для NPL нужен набор компиляторов и других инструментов, отображающих программы NPL на целевые аппаратные платформы. Компилятор FE (препроцессор) отвечает за проверку синтаксиса и семантики программы, а также создаёт промежуточное представление (IR). Компилятор BE отвечает за отображение промежуточных представлений на конкретную аппаратную платформу. Этот компилятор также генерирует интерфейс API, используемый уровнем управления для контроля поведения коммутатора. Компиляторы обеспечивают уровень распараллеливания, определяемый NPL и применяемым оборудованием.

3.1. Преимущества

Разработка NPL началась с обзора имеющихся сетевых платформ и способов предоставления пользователям возможностей программировать их с учётом программных и аппаратных аспектов. Результат требовал возможности раскрыть и применить преимущества базового оборудования. Решение также требовало от конечного пользователя эффективно задать поведение уровня данных и поднять это на уровень ОС и приложений с чётким указанием намерений пользователя. Таким образом, был создан новый язык, позволяющий пользователям задать функциональность сетевого уровня данных с возможностью совместить свойства оборудования с задачами пользователя. Это NPL.

По сравнению с настраиваемыми и другими программируемыми решениями, доступными сегодня, NPL обеспечивает ряд преимуществ. Изощрённые возможности языка обеспечивают:

- настраиваемые конвейеры таблиц;
- интеллектуальное выполнение действий;
- параллельную обработку;
- расширенные возможности логических таблиц;
- уровень интегрированного инструментария;
- простое, интуитивное управление потоками данных.

NPL также поддерживает конструкции, обеспечивающие включение библиотечных компонент, реализующих фиксированные функции аппаратных блоков. Это позволяет описать множество приложений уровня данных с использованием NPL, от простой архитектуры на базе таблиц до сложных систем, включающих множество эффективных блоков. Конструкции языка позволяют выразить эти возможности, что обеспечивает значительный рост эффективности и снижает стоимость финальной аппаратной реализации. Конструкции NPL позволяют многократно использовать программный код при создании семейства устройств коммутации.

3.2. Архитектурная модель

NPL является языком высокого уровня, включающим все компоненты, требуемые для реализации надёжного коммутационного решения. Эти компоненты начинаются с простых типов данных, позволяющих задать отдельные сигналы управления, и включают сложные конструкции высокого уровня для взаимодействия с аппаратными блоками.

На рисунке 1 показана архитектурная модель в виде блок-схемы базовых компонент NPL и связей между ними.

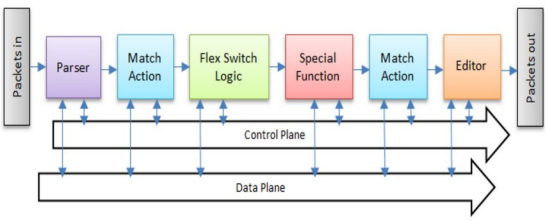


Рисунок 1. Архитектурная модель.

[1]*Network interface card - сетевая интерфейсная плата (адаптер).*

Каждый функциональный блок взаимодействует со своими соседями, читая или записывая данные в одну или несколько шин. Шина содержит набор полей, заданных с использованием NPL. Логически поток шин через блоки образует конвейер обработки. Например. Таблицы «сопоставление-действие» (СД), функции и специальные функции обычно читают поля шины и записывают в них. Блок анализа принимает пакет на входе и записывает значения полей в шину, а блок редактирования использует поля шины для обновления или создания выходного пакета.

Примером такой архитектурной модели может служить последовательность любого числа блоков, расположенных в произвольном порядке[1].

В последующих разделах рассматриваются языковые конструкции, используемые при программировании этих базовых абстракций, с примерами.

4. Компоненты язык NPL

4.1. Поддерживаемые конструкции

Документ поделён в соответствии с функциональными частями конвейера в коммутаторе.

- Типы данных.
- Программные конструкции.
- Конструкции синтаксического анализатора.
- Конструкции шин.
- Конструкции таблиц «сопоставление-действие».
- Функции.
- Конструкции редактора.
- Конструкции специальных функций.
- Метаданные для таблиц «сопоставление-действие» и анализатора.

Определения идентификаторов и констант NPL, а также полная грамматика NPL описаны в Приложении D.

- Идентификаторы должны начинаться с символов [a-z A-Z _] и могут включать символы [a-z A-Z _ 0-9].
- Десятичные и шестнадцатеричные литералы.
- Строковые литералы (например, “foobar”).

В оставшейся части раздела приведены описаний каждой из поддерживаемых конструкций.

4.2. Типы данных

NPL поддерживает базовые типы данных bit, varbit, list, const и auto_enum, а также производный тип struct.

4.2.1. bit

Тип bit относится к базовым. Значение данного типа может быть 0 или 1. Тип служит для описания полей в производных типах данных, таких как struct. Этот тип также применяется в logical_table, logical_register, special_function и других конструкциях.

4.2.1.1. bit-array

Для описания многобитовых полей применяется тип bit-array с указанием размера. NPL не задаёт ограничений для размера битовых массивов. Массивы битовых массивов не поддерживаются в NPL.

Пример

```
bit             cfi;          // однобитовое поле
bit[3]          pri;          // 3-битовое поле pri
bit[12] vid;                  // 12-битовое поле vid
bit[128]        bit_map;      // 128-битовое поле bit_map
bit[8]          label[5];     // не разрешено, поскольку массивы битовых массивов не поддерживаются
```

4.2.1.2. Индексирование bit-array

NPL поддерживает статические и переменные индексы для массивов. Изначально поддержка ограничена битовыми массивами. Размер массива и размер индекса должны соответствовать.

Статическое индексирование bit-array

Индекс задаётся целым числом без знака и может указывать один бит или диапазон битов.

Пример

```
local.rst1 = local.rpa_id_profile1[3:2];
local.rst1 = local.rpa_id_profile1[0:0];
```

Индексирование bit-array переменной

Индексирование массивов переменной обычно применяется для битовых полей (bitmap).

Пример

```
local.rst1 = local.rpa_id_profile1[ip_tmp_bus.idx:ip_tmp_bus.idx];
```

[1]*Полного произвола здесь явно быть не должно, поскольку блоки анализа и редактирования следует размещать в начале и конце конвейера, соответственно. Прим. перев.*

4.2.2. varbit

Тип varbit служит для задания битовых массивов переменного размера. Некоторые протоколы используют в заголовках поля, размер которых может меняться от пакета к пакету. Тип varbit[X] задаёт переменную, размер которой не может превышать X битов.

Пример

```
varbit[120]     options; // опции размером да 120 битов.
```

4.2.3. const

Тип данных const используется для обозначения постоянных величин (integer или enum).

Пример

```
usage_mode_create(in const index,
                      in bit[2] in_pkt_color,
                      in varbit[14] meter_action_set,
                      in varbit[10] color_table_index0,
                      in varbit[8] color_pdd_sbr_index0,
                      out bit[2] color
                     );
```

4.2.4. list

В некоторых конструкциях NPL может требоваться объединение переменного числа элементов в список. Примерами могут служить dynamic_table, strength_resolve и create_checksum. Для представления этого служит тип данных list. Элементы списка должны указываться в фигурных скобках.

```
{ipv4.protocol, ipv4.dip}
```

Списки используются в следующих конструкциях:

- dynamic_table для указания переменного числа входных и выходных полей;
- update_checksum для указания переменного числа, учитываемых в контрольной сумме;
- strength_resolve для указания переменного числа объектов, которые создают запись.

Пример

Аргументы dynamic_table используют list для указания полей, которые могут применяться в заранее выбранном шаблоне.

```
flex_digest_lkup.presel_template(
        {
        ing_cmd_bus.l2_iif_opaque_ctrl_id,
        ing_cmd_bus.vfi_opaque_ctrl_id,
        ing_cmd_bus.l2_iif_flex_digest_ctrl_id_a,
        ing_cmd_bus.l2_iif_flex_digest_ctrl_id_b,
        ing_cmd_bus.fixed_hve_iparser1_0,
        ing_cmd_bus.flex_hve_iparser1_1,
        ing_cmd_bus.fixed_hve_iparser2_0,
        ing_cmd_bus.flex_hve_iparser2_1,
        ing_cmd_bus.my_station_hit
        });
```

Конструкция create_checksum содержит аргумент list со списком полей, учитываемых в контрольной сумме.

```
create_checksum(egress_pkt.fwd_l3_l4_hdr.udp.checksum,
        {
        egress_pkt.fwd_l3_l4_hdr.ipv4.sa,
        egress_pkt.fwd_l3_l4_hdr.ipv4.da,
        editor_dummy_bus.zero_byte,
        egress_pkt.fwd_l3_l4_hdr.ipv4.protocol,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp.src_port,
        egress_pkt.fwd_l3_l4_hdr.udp.dst_port,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp._PAYLOAD
        });
```

4.2.5. struct

Тип struct служит для задания упорядоченного множества полей. Структуры применяются в разных типах конструкций. Внутри структур могут присутствовать лишь типы bit и struct. Тип struct поддерживает перекрытия (overlay) для указания полей несколькими способами

Таблица 1. Конструкция struct.

Конструкция	***Аргументы, опции***	***Описание***
struct		Задаёт новую структуру с именем и полями.
	fields	bit, bit[n], varbit или struct. Другие типы и конструкции не разрешены.
	overlays	Задаёт наложения для полей структуры. В структуре разрешается лишь одна конструкция overlays. Все наложения struct указываются в конструкции overlays.

4.2.6. Массивы struct

В NPL разрешены одномерные массивы struct. NPL не ограничивает размер массива struct. Ниже приведены примеры использования массивов struct.

```
obj_bus.struct1[arr1].field = field;
obj_bus.struct1[arr1].struct2[arr2].field = field;
```

```
cmd_bus.struct1 = obj_bus.struct1[arr];
```

Пример

Простая структура

```
struct vlan_s {
        fields {
                bit             cfi;    // 1-битовое поле
                bit[3]          pri;    // 3-битовое поле pri
                bit[12] vid;    // 12-битовое поле vid
        }
}
```

Структура с наложением

```
struct switch_bus_s {
        fields {
                bit[4]  otpid_enable;
                bit     olp_enable;
                bit     ts_enable;
                bit[10] ing_port_num; // Базовое поле для определённых далее наложений.
                bit     svp_enable;
        }
        overlays {
                ing_svp :       ing_port_num[7:0];
                ing_pri :       ing_port_num[9:8]; // Наложения на базовое поле ing_port_num.
                exp :           ing_port_num[9:8];
        }
}
```

Массив структур

```
struct mpls_header_stack_t {
        fields {
                mpls_t mpls[3]; // Здесь может быть 3 заголовка mpls_t.
        }
}
```

Элементы массива можно указать в форме mpls[0], mpls[1], mpls[2].

4.2.7. enum

NPL поддерживает конструкцию enum для определения перечисляемых типов. Значения элементов enum должен предоставлять пользователь. Перечисляемое в NPL - это просто идентификатор, указывающий подмножество того, что предоставляется в C/C++, и не является типом данных в NPL. Перечисляемые служат для представления констант и определяют константы, используемые в качестве аргументов функций и rvalue в операторах присваивания.

Пример

```
enum drop_reason{
        NO_DROP = 0,
        MEMBERSHIP_DROP = 1,
        TTL_DROP = 2
}
packet_drop(drop_bus.disable_drop, drop_reason.TTL_DROP, 5);
```

4.2.8. auto_enum

NPL поддерживает тип данных auto_enum для задания перечисляемых типов. Значения элементов auto_enum назначает компилятор. Производитель целевой платформы может принять решение о способе отображения и присваивания значений auto_enum.

Типовыми применениями auto_enum являются Logical Table Lookup, Multi-Data View и Strength Based Resolution Index.

Целевой компилятор может задавать значения auto_enum на основе контекста их использования. Такие auto_enum должны быть глобальными и назначаться в одном экземпляре. Например, auto_enum из поиска в логических таблицах нельзя использовать в специальных функциях.

Пример

```
auto_enum qos_entry {
        QOS_DISABLE,
        QOS_L3_TUNNEL,
        QOS_L2_TUNNEL
}
qos_sfc.sf_profile_entry("sfc_qos_profile", qos_entry.QOS_L3_TUNNEL,
        {
                obj_bus.mapping_ptr,
                cmd_bus.effective_exp
        },
        {
                cmd_bus.int_pri,
                cmd_bus.pri
        }
);
```

4.3. Выражения

4.3.1. Обозначение чисел

NPL поддерживает только десятичные и шестнадцатеричные литеральные константы. Числовая нотация применяется при задании значений полей. NPL не поддерживает тип bool, значение 0 соответствует false, все прочие - true.

Пример

```
a = 5;                  // десятичное значение
ipv4.ttl = 0xF;         // шестнадцатеричное значение
ipv6.dip = 0x01234567;
ipv6.dip[63:0] = 0x0123456789abcdef;.
if (ipv4.protocol == 0x23)
ipv4.protocol = 0x231;
```

4.3.2. Условные операторы

NPL поддерживает операторы условий в разных конструкциях.

Таблица 2. Условные конструкции NPL.

Условные операторы	***Описание***
if, else if, else	Оператор if
switch	Оператор switch

4.3.3. Операторы

NPL поддерживает множество операторов. Ограничения при их использовании рассмотрены ниже.

Таблица 3. Операторы NPL.

Оператор	***Символ***	***Описание***
Арифметические операторы	+	Сложение
	-	Вычитание
	*	Умножение
	/	Деление
	%	Деление по модулю
Операторы отношений	==	Равно
	!=	Не равно
	<	Меньше
	<=	Меньше или равно
	>	Больше
	>=	Больше или равно
Оператор слияния	<>	Конкатенация
Логические операторы	&&	Логическое И (AND)
	||	Логическое ИЛИ (OR)
Операторы сдвига	<<	Сдвиг влево
	>>	Сдвиг вправо
Побитовые логические операторы	&	И
	|	ИЛИ
	!A	Отрицание (NOT)
	~A	Дополнение до 1
	^	Исключающее ИЛИ (XOR)
Унарные операторы	&A	Сокращение И (все биты 1)
	|A	Сокращение ИЛИ (все биты 0)
Оператор присваивания	=	Присваивание значений[1]
Оператор маскирования	mask	Применяется в операторе switch, трактуется как AND. Операнды слева и справа от = могут иметь разные размеры. Если левый операнд (lvalue) больше правого (rvalue) по размеру значение дополняется нулями. В противном случае компилятор возвращает ошибку.

4.3.4. Области действия переменных

4.3.4.1. Глобальные переменные

Ниже приведён список переменных NPL с глобальной областью действия. Такие переменные должны иметь уникальные имена:

- enum;
- auto_enum;
- struct;
- bus;
- packet;
- logical_table;
- logical_register;
- parser_node;
- function;
- special_function;
- dynamic_table;
- strength.

[1] *Здесь явно следовало бы указать дополнение нулями слева или справа. Текущая формулировка может приводить к ошибкам в реализации. Прим. перев.*

4.3.4.2. Локальные переменные

Во многих конструкция используются переменные с локальной областью действия.

- logical_table - поля, ключи;
- struct - поля, структуры;
- logical_register - поля;
- enum - элементы;
- auto_enum - элементы;
- special_function - методы;
- dynamic_table - методы.

4.4. Конструкция program

Программы представляют собой приложения NPL. Имя программы является точкой входа, как main в языке C. Конструкция program определяет порядок выполнения других конструкций конвейера обработки пакетов. Для управления потоком обработки по условиям используются конструкции if-then-else.

Программа может вызывать перечисленные ниже конструкции с помощью ключевых слов, указанных в таблице 4.

- дерево синтаксического анализатора;
- поиск в таблицах (logical_table, dynamic_table);
- функции обработки пакетов;
- специальные функции;
- сравнение силы.

Присваивание значений не допускается в конструкции program.

Таблица 4. Конструкция program (порядок выполнения).

Конструкция	***Аргументы, опции***	***Описание***
program		Задаёт порядок выполнения операций обработки пакета.
	Условные конструкции	В конструкциях с условием внутри program можно вызывать: - конструкции и операторы if/else/else if и switch; - все операторы NPL.
	parse_begin(<имя узла>)	Прохождение дерева синтаксического анализа от корневого узла. Для разных пакетов могут применяться разные деревья.
	parse_continue(<имя узла>)	Возврат процесса к заданному узлу дерева анализа.
	<table>.lookup(<lookup_num>)	Поиск в таблице с автоматическим выполнением связанных с таблицей методов.
	Вызовы функций обработки пакетов	
	Вызовы специальных функций	
	Сравнение силы	Выбор результата при поиске в нескольких таблицах.

Пример

```
program mim_main () {
        parse_begin (ethernet); /* Начало анализа из узла parser_node Ethernet. */
        port.lookup (0);                /* Поиск в таблице port. */
        iif.lookup (0);         /* Поиск в таблице iif. */
        my_station.lookup (0);  /* Поиск в таблице my_station. */
        isid.lookup (0);                /* Поиск в таблице isid. */
        mim_isid_switch_logic1();       /* Выполнение логической функции. */
        ...
        if (cmd_bus.do_l3) {    /* Поиск по условию. */
                l3_host.lookup (0);
        }
        ...
        l3_switch_logic1();             /* Вызов функции обработки пакета. */
        ...
        next_hop.lookup (0);    /* Поиск в таблице next_hop. */
        do_packet_edits();              /* Функция редактирования. */
}
```

4.5. Конструкция синтаксического анализатора

Конструкция parser определяет:

- заголовки - упорядоченный набор полей фиксированного или переменного размера;
- группы заголовков;
- пакеты, состоящие из групп заголовков;
- связи между типами заголовков, формирующие дерево анализа.

NPL позволяет задавать базовые типы заголовков с помощью struct. Спецификация анализатора использует типы заголовков при объявлении struct для групп заголовков, а группы - при объявлении struct для пакетов. Программы NPL должны определять пакеты в форме packet.header_group.header.

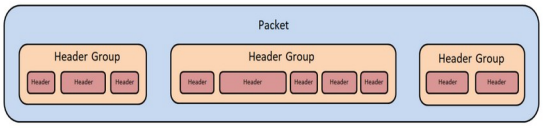


Рисунок 2. Заголовки, группы и пакеты.

4.5.1. Заголовок (struct)

Заголовки определяются с помощью типа данных struct, в которых поля заголовка определяются типами bit, bit-array и varbit для задания полей переменного размера, где header_length_exp служит для указания размера поля varbit. В NPL поддерживаются массивы заголовков.

Таблица 5. Конструкция заголовка (struct).

Аргументы, опции	***Описание***
struct	Задаёт новый тип заголовка.
varbit	При наличии полей переменного размера они задаются типом varbit с указанием максимального размера. В заголовке допускается лишь одно поле типа varbit, которое должно указываться последним. NPL не ограничивает размер таких полей.
header_length_exp	Для полей переменного размера задаёт выражение, определяющее реальный размер. В выражении можно использовать операторы + и *. Выражения следует указывать в форме var * c0 + c1, где var является полем заголовка, а c0 и c1 - константами. Для полей фиксированного размера это поле не требуется.

Пример

Статический заголовок

```
struct vlan_t {
        fields {
                bit[3]          pcp;
                bit             cfi;
                bit[12] vid;
                bit[16] ethertype;
        }
}
```

Задаёт размер и порядок упаковки полей в заголовок пакета.

Заголовок переменного размера

```
struct ipv4_t {
        fields {
        bit[4]  version;
        bit[4]  hdr_len;
        bit[8]  tos;
        bit[32] sa;
        bit[32] da;
        varbit[320]     option; // максимальный размер поля
}
header_length_exp:      hdr_len*4;      // задаёт способ расчёта числа байтов в заголовке
```

В приведённом примере option является полем переменного размера, а header_length_exp указывает способ вычисления размера заголовка. Другим примером может служить header_length_exp : (payload_len*4)+2. Вы выражении допускаются операторы + и *. В varbit[num] значение num указывает максимальный размер поля. Для каждого заголовка с полями переменного размера должен указываться атрибут header_length_exp. Структуры с несколькими полями переменного размера не поддерживаются.

4.5.2. Группа заголовков (struct)

Группы заголовков определяются с помощью типа struct и впоследствии служат для задания пакетов (4.5.3. Конструкция для пакета). Специальных опций для таких структур не поддерживается. Структура уровня группы заголовков позволяет «массовые» манипуляции и ссылки на заголовки в пакетах. Заголовки помещаются в группу в порядке, указанном NPL.

- Группа заголовков может включать лишь struct с заголовками. Типы bit или bit-array не разрешаются.
- Массивы групп заголовков не поддерживаются.
- Заголовки и группы заголовков должны указываться в порядке их размещения в пакетах. Это важно!

Пример

Группа заголовков с множеством struct

```
struct l2_header_t {
        fields {
                bit[48] macda;
                bit[48] macsa;
                bit[16] ethertype;
        }
}

struct vlan_tag_t {
        fields {
                bit[3]          pcp;
                bit             cfi;
                bit[12] vid;
```

```
        }
}

struct group0_t {
        fields {
                l2_header_t     l2_header;
                vlan_tag_t      ovlan;
        }
}
```

Для группы заголовков struct задаёт порядок размещения заголовков в группе.

Задание массива заголовков

```
struct group1_t {
        fields {
                mpls_t mpls[3]; // говорит о наличии трёх заголовков mpls_t.
        }
}
```

Элементы массива можно указывать в форме mpls[0], mpls[1], mpls[2].

4.5.3. Конструкция для пакета

Пакет состоит из групп заголовков и в NPL представляется с помощью struct. Экземпляры пакета объявляются с ключевым словом packet

```
packet struct-name instance-name;
```

Здесь объявляется пакет с именем instance-name, который описан в структуре struct-name, называемой структурой уровня пакета. Элементами этой структуры должны быть структуры групп заголовков. Структура для пакета не может содержать типы bit и bit-array, а экземпляры пакетов не могут быть массивами.

Структура уровня пакета служит для объединения групп заголовков в пакет.

Пример

Определение структуры и создание экземпляра пакета

```
struct macs_t { // Структура заголовка, где все элементы являются массивами битов
        fields {
                bit[48] dmac;
                bit[48] smac;
        }
}

struct vlan_t { // Структура заголовка
        fields {
                bit[16] tpid;
                bit[3]          pcp;
                bit             dei;
                bit[12] vid;
        }
}

struct ethertype_t {    // Структура заголовка
        fields {
                bit[16] type;
        }
}

struct mpls_t { // Структура заголовка, где все элементы являются массивами битов
        fields {
                bit[20]         label;
                bit[3]  tc;
                bit             s;
                bit[8]  ttl;
        }
}

struct mpls_grp_t {     // Структура группы заголовков
        fields{
                mpls_t mpls[3];
        }
}

struct ipv4_t { // Структура заголовка
        fields {
                // Определения полей IPv4 (только bit-array)...
        }
}

struct ipv6_t { // Структура заголовка
        fields {
                // Определения полей IPv6 (только bit-array)...
        }
}

struct l2_t {   // Структура группы заголовков, где все элементы являются структурами
```

```
        fields {
                macs_t  macs;
                vlan_t  ctag;
                ethertype_t     etype;
        }
}

struct l3_t {   // Структура группы заголовков
        fields {
                ipv4_t ipv4;
                ipv6_t ipv6;
        }
}

struct ingress_packet_t {       // Структура уровня пакета
        fields {
                l2_t            l2;
                mpls_grp_t      mpls_grp;
                l3_t            l3;
        }
}
packet ingress_packet_t ing_pkt;        // Указывает ingress_packet_t как структуру уровня пакета
```

Заголовок и поля в пакете должны указываться, как приведено ниже.

```
ing_pkt.l2.macs.da
```

Ниже показано, как должны задаваться массивы в заголовке.

```
ing_pkt.mpls_grp.mpls[0].label
```

Платформы могут вносить ограничения для доступа и изменения пакетов. Например, для разделения архитектуры Ingress и Egress могут использоваться входные и выходные пакеты, при этом запись во входные может быть отключена. Все изменения могут вноситься лишь в выходные пакеты.

4.5.4. Метаданные заголовка

С каждым заголовком связано 1 битовое поле метаданных _PRESENT, указывающее, пригоден ли конкретный экземпляр заголовка для текущего пакета. Это делает метаданные _PRESENT динамическими. При наличии заголовка во входящем пакете для флага _PRESENT устанавливается значение 1. Поле _PRESENT доступно лишь для чтения и сохраняется в течение срока жизни пакета.

Пример

```
struct tcp_t {
        fields {
                ...
        }
}

struct group0_t {
        fields {
                tcp_t tcp;
                ...
        }
}

struct packet_t {
        fields {
                group0_t group0;
        ...
        }
}

packet packet_t ing_pkt;

program l3 () {
        ...
        if (ing_pkt.group0.tcp._PRESENT) {
                l2_table.lookup(0);
        }
        ...
}
```

4.5.5. Соединения дерева анализа (parser_node)

Конструкция parser_node задаёт соединения экземпляров заголовков в пакетах. По сути, она определяет узлы анализа и переходы. В parser_node поддерживаются условные конструкции для перехода к следующему parser_node.

Таблица 6. Конструкция parser_node.

Конструкция	***Аргументы, опции***	***Описание***
parser_node		Задаёт узел синтаксического анализатора и его соединения со следующими узлами.
	name	Имя parser_node.
	root_node	В дереве анализатора может быть лишь один корневой узел. Функция parse_begin() может указывать только один корневой узел. Для прерывания и повторного входа в дерево анализа служат parse_break и parse_continue.
	next_node<node>	Задаёт следующий узел, с которым соединён данный parser_node.

switch	Задаёт оператор switch для описания условия перехода к следующему parser_node. В операторах может применяться ключевое слово mask. В операторах разрешены битовые массивы. Значения вариантов могут быть константами или константами с маской (операция AND).
if/else	Условие перехода к следующему узлу. Поддерживаются операторы сравнения == и !=, если rvalue является константой. Поддерживаются логические операции &&, ||, !.
extract_fields(packet.header)	Задаёт экземпляр заголовка для анализа. Текущая позиция анализа пакета выходит за пределы этого заголовка.
parse_break(<node>)	Задаёт выход из дерева анализа и возврат в программу. Указанный аргументом узел является следующим узлом, с которого возобновляется анализ при возврате в дерево.
end_node	Указывает последний лист в дереве.
latest	Указывает последний заголовок, проанализированный в этом parser_node. Имя latest.field может использоваться в дереве анализа.
current	Указывает текущие байты пакета для выравнивания. Например, для просмотра следующих 2 байтов в пакете следует использовать current. При вызове extract_fields указатель current выходит за пределы анализируемого заголовка. `current (смещение первого бита, число выбираемых битов)`
default	Задаёт в операторе switch вариант, применяемый при отсутствии иного подходящего.

Пример

Задание дерева анализа

```
struct vlan_t {
        fields {
                bit[3]          pcp;
                bit[1]          cfi;
                bit[12] vid;
                bit[16] ethertype;
        }
}

struct l2_t {
        fields {
                bit[48] macda;
                bit[48] macsa;
                bit[16] ethertype;
        }
}

struct group1_t {
        fields {
                l2_t    l2;
                vlan_t  vlan;
        }
}

struct ing_pkt_t {
        fields {
                group1_t group1;
        }
}

parser_node start {
        root_node : 1;
        next_node ethernet;
}

parser_node ethernet {
        extract_fields(ing_pkt.group1.l2);
        switch (latest.ethertype) {
                0x8100          : {next_node ctag};
                default : {next_node ingress};
        }
}

parser_node ctag {
        extract_fields(ing_pkt.group1.vlan);
        if (current(0,16) == r_ing_outer_tpid_0.tpid) {
                next_node otag;
        }
        next_node ingress;
}

parser_node ingress {
        end_node : 1;
}
```

4.5.6. Выход и повторный вход в дерево (parse_break, parse_continue)

NPL поддерживает механизм для поиска в таблицах и выполнения иной работы с пакетами в процессе анализа заголовков. Это может требоваться в тех случаях, когда решению в узле анализатора нужен результат поиска в таблице. Поток анализатора прерывается с возвратом управления в программу, а после выполнения требуемых действий возвращается в тот же узел. Для этого служат конструкции parse_break и parse_continue.

Пример

Выполнение поиска в таблице перед анализом пакета ethernet и mpls.

```
program mpls_switch() {
        parse_begin(start);
        port_table.lookup(0);
        // узел ethernet принимает вывод из port_table, т. е. logical_bus.otpid_enable
        parse_continue(ethernet);
        // узел mpls_label принимает вывод из mpls_table, т. е.
        // logical_bus.mpls_table_result_type
        mpls_table.lookup(0);
        parse_continue(mpls_label);
}
parser_node start {
        root_node : 1;
        switch(logical_bus.rx_port_parse_ctrl) {
                0x0:    next_node ppd;
                0x2:    next_node sobmh;
                0x3:    parse_break(ethernet);
                default: next_node ingress;
        }
}
parser_node ethernet {
        extract_fields(ingress_pkt.outer_l2_hdr.l2);
        if (logical_bus.otpid_enable[3:3] && latest.ethertype == 0x8100) {next_node otag;}
        //0x8100
        if (logical_bus.otpid_enable[2:2] && latest.ethertype == 0x8100) {next_node otag;}
        //0x8100
        if (logical_bus.otpid_enable[1:1] && latest.ethertype == 0x8100) {next_node otag;}
        //0x8100
        if (logical_bus.otpid_enable[0:0] && latest.ethertype == 0x8100) {next_node otag;}
        //0x8100
}
parser_node mpls_0 {
        extract_fields(ingress_pkt.outer_l3_l4_hdr.mpls[0]);
        switch (latest.stack) {
                0x0: next_node mpls_1;
                0x1: parse_break(mpls_label);
                default: next_node ingress;
        }
}
parser_node mpls_label {
        extract_fields(ingress_pkt.outer_l3_l4_hdr.mpls[4]);
        switch (logical_bus.mpls_table_result_type) {
                0x0: next_node mpls_cw;
                0x1: next_node inner_ethernet;
                0x2: next_node inner_l3_speculative;
                default: next_node ingress;
        }
}
parser_node ingress {
        end_node:1;
}
```

Запись end_node:1 указывает завершение анализа.

4.6. Конструкция логической шины

Конструкции логических шин служат для определения набора полей (переменных)).

4.6.1. Определение шины

Шины создаются с помощью конструкции struct, содержащей поля и наложения. Указанный в структуре порядок полей поддерживается логической шиной.

4.6.2. Создание экземпляра шины (bus)

Для создания шины используется ключевое слово bus, а шина определяется с помощью struct. Шина может включать наложение полей, которые могут индивидуально указываться в программе NPL.

Пример

Создание экземпляра шины

```
struct control_bus_t {
        fields {
                bit     ts_enable;
                bit     olp_enable;
                bit[4] otpid_enable;
        }
```

```
}

bus     control_bus_t   control_id;

parser_node pkt_start{
        root_node : 1;
        next_node ethernet;
}

parser_node ethernet {
        extract_fields(ing_pkt.group0.l2);
        if (control_id.ts_enable == 0) { // control_id - логическая шина,
                                         // ts_enable - поле шины.
                if (control_id.otpid_enable != 0 ) {
                        switch (latest.ethertype) {
                                0xABCD  : {next_node vntag};
                                0x8888  : {next_node etag};
                                0x8100  : {next_node otag};
                                0x9100  : {next_node itag};
                                0x0000 mask 0xFC00      : {next_node llc};
                                default : {next_node payload};
                        }
                }
        }
}

parser_node otag{
        extract_fields(ing_pkt.group0.ovlan);
        end_node:1;
}
```

4.7. Конструкции логических таблиц

Конструкция логической таблицы служит для определения таблицы с ключами (keys), полями fields, key_construct, fields_assign, а также minsize и maxsize. Таблицы также имеют встроенный метод lookup().

4.7.1. Логическая таблица (logical_table)

Конструкция logical_table служит для объявления таблиц «сопоставление-действие» (СД). Это позволяет пользователю задать структур данных, которую уровень управление или уровень данных может менять. Пользователь может задать поля ключа и правила для хранения в таблице, а также механизм создания ключей с использованием полей логической шины. Кроме того, logical_table позволяет задать метод fields_assign для работы с полями.

Ключи и поля логической таблицы имеют локальную значимость. Поиск в таблицах NPL может выполняться многократно, в зависимости от возможностей целевой архитектуры.

Все объявленные логические таблицы должны вызваться конструкцией вида <имя таблицы>.lookup(lookup_num). lookup_num = 0 указывает первый поиск.

Таблица 7. Конструкция logical_table.

Конструкция	***Аргументы, опции***	***Описание***
logical_table		Задаёт новую таблицу.
	table_name	Задаёт имя таблицы.
	table_type	Задаёт тип таблицы с точки зрения пользователя. Допустимы типы index, tcam, hash, alpm. Компилятор может отображать типы на разные компоненты, а целевая платформа может добавлять свои типы.
	keys	Задаёт ключи, используемые для доступа к логической таблице. Размер ключа задаётся объявлением bit. Ключи могут иметь тип bit или bit-array, тип struct не разрешён для ключей.
	fields	Задаёт поля правил для logical_table. Размер поля задаётся объявлением bit. Поля могут иметь тип bit, bit-array и auto_enum, тип struct не разрешён. Тип auto_enum служит для задания множества представления данных.
	key_construct()	Задаёт логику создания ключей таблицы с использованием приведённых ниже правил. - Может поддерживаться множество поисков в одной таблице (4.7.3. Множественный поиск в таблице). - Для множественного поиска разрешены условные выражения с метаданными _LOOKUP0 и _LOOKUP1 и т. д. - Ключи создаются из полей шин.
	fields_assign()	Метод описания функциональности обработки и назначения полей логической шины. - Может поддерживаться множество поисков в одной таблице (4.7.3. Множественный поиск в таблице). - Для множественного поиска в данной таблице разрешены условные выражения с метаданными _LOOKUP0 и _LOOKUP1 и т. д. - Назначения могут дополнительно ограничены _VALID и multi-view(auto_enum) - Другие условия и операции не разрешены в блоках fields_assign.

minsize	Минимальный гарантированный размер. Физическая таблица должна иметь такое число элементов.
maxsize	Максимальный разрешенный размер. При разных maxsize и minsize это значение служит в основном для заполнения SDK[1]. Одинаковые значения minsize и maxsize считаются «размером» таблицы и компилятор должен найти физическую таблицу указанного размера.

Пример

Определение индексной таблицы

```
logical_table port {
        table_type : index;
        minsize : 128;
        maxsize : 128;
        keys {
                bit[7] port_num;
        }
        fields {
                bit[1] l3_enable;
                bit[1] otag_enable;
                bit[8] src_modid;
                bit[12] default_vid;
        }
        key_construct() {
                port_num = obj_bus.port_num;
        }
        fields_assign() {
                if (_LOOKUP0 == 1) {
                        cmd_bus.port_l3_enable = l3_enable;
                        ...
                }
        }
}
```

Определение таблицы TCAM

```
logical_table my_station_hit {
        table_type : tcam;
        maxsize : 512;
        minsize : 512;
        keys {
                bit[48] macda;
                bit[12] vid;
                bit[8] src_modid;
        }
        fields {
                bit[2] mpls_tunnel_type;
                bit    local_l3_host;
        }
        key_construct() {
                macda   = ing_pkt.l2_grp.l2.macda;
                vid     = obj_bus.vlan_id;
                src_modid = obj_bus.source_logical_port;
        }
        fields_assign() {
                if (_LOOKUP0 == 1) {
                        l3_cmd_bus.local_l3_host = local_l3_host;
                        ...
                }
        }
}
```

Вызов таблицы

```
program ingress {
        port.lookup(0); //calls port
        logical table
        if (cmd_bus.vlan_valid == 1) {
                my_station_hit.lookup(0);
                // Вызов поиска в my_station_hit первый раз
                my_station_hit.lookup(1);
                // Вызов поиска в my_station_hit второй раз
        }
}
```

4.7.2. Метаданные логической таблицы

В NPL каждая логическая таблица имеет перечисленные ниже метаданные. Для каждого пакета значение метаданных присваивается с использованием ряда правил.

- _LOOKUPx - 1-битовое значение, устанавливается при поиске в таблице для пакета.
- _HIT_INDEXx - 32-битовое значение, указывающее строку таблицы, которой соответствует пакет. Формат _HIT_INDEXx может зависеть от целевой платформы. Должен использоваться один бит, показывающий, соответствует ли поиск действительной записи.
- _VALID - 1-битовое значение, устанавливаемое если поиск даёт действительную запись.

[1] *Software Development Kit - пакет для разработки программ.*

В именах x представляет lookup_num (0, 1 и т. д.).

Пример

```
logical_table table_a {
        ...
        fields_assign() {
                if (_LOOKUP0) {
                        obj_bus.src_hit_index = _HIT_INDEX0;
                }
                if (_LOOKUP1) {
                        obj_bus.dst_hit_index = _HIT_INDEX1;
                }
        }
}
```

Как и метаданные заголовка, это повышает удобочитаемость и обеспечивает основу для инструментария.

4.7.3. Множественный поиск в таблице

NPL позволяет задать множественный поиск в одной logical_table. В этом случае могут применяться метаданные _LOOKUP0, _LOOKUP1 и т. д., чтобы различать ключи и поля, обрабатываемые в блоках key_construct() и fields_assign().

Пример

```
//Определение логической таблицы
logical_table mac_table {
        table_type : hash;
        minsize : 64;
        maxsize : 64;
        keys {
                bit[48] macda;
        }
        fields {
                bit[16] port;
                bit[1] dst_discard;
                bit[1] src_discard;
        }
        key_construct() {
                if (_LOOKUP0==1) {
                        macda = ing_pkt.l2_grp.l2.da;
                }
                if (_LOOKUP1==1) {
                        macda = ing_pkt.l2_grp.l2.sa;
                }
        }
        fields_assign() {
                if (_LOOKUP0==1) { // Например, Entry 100
                        obj_bus.dst = port;
                        obj_bus.dst_discard = dst_discard;
                }
                if (_LOOKUP1==1) { //Например, Entry 200
                        temp_bus.src_port = port;
                        obj_bus.src_discard = src_discard;
                }
        }
}
program {
        if ((ing_pkt.l2_grp.l2._PRESENT) & (ing_pkt.l2_grp.vlan.vid != 0)) {
                // Условие поддерживается.
                mac_table.lookup(0);
                mac_table.lookup(1);
        }
}
```

4.7.4. Множество типов данных (режимы размера данных)

Внутри логических таблиц поля могут упаковываться в разные форматы, которые могут требоваться по причинам размера или наложения разных данных. Это применяется для повышения эффективности.

Разработчик NPL должен задать все эти поля разных типов данных в конструкции fields{}. Например, логическая таблица NHI имеет два представления данных:

представление 1 - поля A, B, C;

представление 2 - поля A, D, E, F.

Правила NPL _VALID:

- если логическая таблица имеет много типов данных, она будет включать 1 вхождение _VALID (_VALID = 0);
- если некоторые поля имеют strength, они должны быть указаны в разделе _VALID=0 внутри блока fields_assign(). Вызовы strength выполняются лишь для случаев _VALID=1.

Пример

```
auto_enum multi_view {
        UC_VIEW,
        MC_VIEW,
```

```
        BC_VIEW
}
logical_table NHI {
        ...
        fields {
                bit[3]  A;
                bit[15]         B;
                bit[7]  C;
                bit[10]         D;
                bit[4]  E,
                bit[4]  F;
                bit[16]         strength_object_G;
                multi_view X; // поле data_type для обозначения разных представления (auto_enum).
        }
        fields_assign() {
                if (_LOOKUP0 == 0) {
                        if (_VALID == 1) { // _VALID – то же, что «попадание».
                                if (X == UC_VIEW) {
                                        bus.A = A;
                                        bus.B = B;
                                        bus.C = C;
                                }
                                if (X == MC_VIEW) {
                                        bus.A = A;
                                        bus.D = D;
                                        bus.E = E;
                                        bus.F = F;
                                }
                        } // завершение _VALID == 1
                        else { // _VALID == 0 задаёт лишь поля data_type = 0.
                                bus.A = 0;
                                bus.B = 0;
                                bus.C = 5;      // пример ненулевой константы.
                                bus.G = 0;
                        } // завершение _VALID == 0
                }
                if (_LOOKUP1 == 1) {
                        ...
                }
        }
}
```

4.8. Конструкция логического регистра

Конструкция logical_register служит для задания объекта со множеством полей и глубиной 1 (регистр). Логический регистр обеспечивает программам интерфейс для настройки элементов управления. В отличие от таблиц регистры не имеют ключей поиска. Поля могут инициализироваться во время компиляции и заполняться в процессе работы.

4.8.1. Определение одноуровневого хранилища

Конструкция logical_register задаёт один логический регистр со множеством полей. Результат всегда доступен для вызывающей функции, индекс не требуется. Логические регистры не могут использоваться для поддержки состояний, связанных с разными пакетами. Эти регистры содержат лишь конфигурацию уровня управления.

Регистры часто полезны в деревьях анализа, функциях и т. п. Они могут применяться в качестве констант, настраиваемых уровнем управления.

Таблица 8. Конструкция logical_register.

Конструкция Аргументы, опции	***Описание***
logical_register	Задаёт новый регистр, который может иметь произвольный размер.
fields	Задаёт поля данных логического регистра. Размер полей указывается с использованием типа bit. Для каждого поля должно указываться значение при сбросе.

Пример

Определение логического регистра

```
// Эта конструкция может использоваться, например, для задания значений, подобных TPID.
logical_register tpid_values {
        fields {
                bit[16] tpid0 = 0x8100;
                bit[16] tpid1 = 0x9100;
                bit[16] tpid2 = 0x7100;
                bit[16] tpid3 = 0x8868;
        }
}
```

В определении указан размер и значение каждого поля при сбросе.

4.9. Функции обработки пакетов (function)

Функции применяются в NPL для описания базовой обработки пакетов и обработки результатов синтаксического анализа, логических таблиц, special_function и других конструкций. Функции являются императивными конструкциями, которые могут преобразовывать данные и поддерживать модульность приложений. Из функций могут вызываться другие конструкции NPL.

Функции поддерживают условные операторы, операторы присваивания и комплексные операции преобразования данных на логических шинах. Для функций поддерживается вложенность. Объявления конструкций внутри функции не разрешаются.

Имеется несколько сценариев использования функций и это позволяет реализовать гибкую логику принятия решений. Например, можно декодировать результаты поиска для идентификации индивидуальных (unicast) и групповых (multicast) пакетов. Функции можно применять для извлечения данных из пакета, вызова поиска в логических таблицах, выполнения специальных функций.

Функции позволяют также организовать модульную структуру приложения. Для этого функция может включать поиск в логических таблицах, сравнение силы, вызовы динамических таблиц и т. п. Все элементы, которые могут быть заданы в конструкции program, подходят для функций. Рекомендуется создавать отдельные функции для поддержки модульности и обработки пакетов.

Целевые платформы могут ограничивать область действия и применение функций с учётом аппаратных возможностей.

Таблица 9. Конструкция function.

Конструкция Аргументы, опции	***Описание***
function	Задаёт новую функцию обработки пакетов.
function_name	Имя функции.
	Любые условные, арифметические и логические операторы, манипуляции с логическими шинами, поиск в logical_table, special_function, editor, strength.

Пример

```
// Шина будет применяться внутри функций.
struct switch_logic_t {
        fields {
                bit no_l3_switch;
                bit l2_same_port_drop;
        }
}

// Логические регистры доступны из функций.
logical_register cpu_control {
        fields {
                bit
                tunnel_to_cpu = 0; // Инициализация.
        }
}

bus switch_logic_t temp;

function l3_switch_logic1 () {
        temp.no_l3_switch = 0;
        if (port.l3_enable &&
                (ingress_pkt.outer_l3_l4_hdr.ipv4._PRESENT ||
                ingress_pkt.outer_l3_l4_hdr.ipv6._PRESENT)
           ) {
                if (obj_bus.tunnel_pkt || obj_bus.tunnel_error) {
                        if (obj_bus.tunnel_error) {
                                obj_bus.tunnel_decap = 0;
                                temp.no_l3_switch = 1;
                                if (cpu_control.tunnel_to_cpu) {
                                        obj_bus.copy_to_cpu = 1;
                                }
                        } else {
                                obj_bus.tunnel_decap = 1;
                        }
                } else { // Не туннельный пакет.
                        obj_bus.tunnel_decap = 0;
                }
        }
        // Трассировка пакета
        packet_trace(temp.no_l3_switch, cpu_reason.NO_SWITCH);

        temp.l2_same_port_drop = obj_bus.src_prune_en && (obj_bus.l2_oif == obj_bus.l2_iif);
        // Отбрасывание пакета
        packet_drop(temp.l2_same_port_drop, drop_reason.L2_SAME_PORT_DROP,
                        L2_SAME_PORT_DROP_STR);
}
```

4.10. Конструкции редактирования пакетов

Конструкции для редактирования пакетов включают добавление нового заголовка (поля создаются с использованием функции), удаления и изменения заголовка (в логической шине). Изменённый пакет (после редактирования) должен соответствовать одному из описанных в графе анализа пакетов, поэтому в редакторе нужно использовать имена из дерева синтаксического анализа.

Создание нового заголовка не входит в конструкцию редактора, которая должна вызываться из функции. Входные пакеты открыты лишь для записи и не могут редактироваться - все операции редактирования выполняются с выходными пакетами.

4.10.1. Добавление заголовка

Новый заголовок создаётся с использованием функций, после чего добавляется в пакет. Компилятор редактора распознает заголовок и будет работать с ним.

Таблица 10. Конструкция add_header.

Конструкция	Аргументы, опции	Описание
add_header		Задаёт добавление заголовка в пакет.
	new_header_name	Имя нового заголовка, совпадающее с именем в спецификации пакета.

Пример

Добавление заголовка в пакет

Если приложению нужно добавить otag и создать его до вызова add_header(otag), можно задать

```
egr_pkt.group1.otag.vid = ing_pkt.itag.vid+100; // otag и itag заданы как заголовки в packet.
egr_pkt.group1.otag.pcp = obj_bus.egr_port_table_pcp; // из шины object.
egr_pkt.group1.otag.tpid = 0x9100;
add_header(egr_pkt.group1.otag);
```

Добавление туннельного заголовка к пакету

Если приложению нужно добавить туннельный заголовок Tunnel L2 и VLAN ID, можно задать

```
egr_pkt.group2.tunnel_l2.dmac = 0xff;
egr_pkt.group2.tunnel_l2.smac = obj_bus.l3_interface_smac;
egr_pkt.group2.tunnel_l2.vid = obj_bus.l3_next_hop_vid;
add_header(egr_pkt.group2.tunnel_l2);
```

4.10.2. Удаление заголовка

Удаляет заголовок из стека заголовков пакета. Применяется в некоторых приложениях, таких как выход краевого коммутатора, для удаления туннельных заголовков.

Таблица 11. Конструкция delete_header.

Конструкция	Аргументы, опции	Описание
delete_header		Задаёт удаление заголовка из пакета. Для указания заголовка используется спецификация синтаксического анализа.
	header_name	Имя удаляемого заголовка, совпадающее с именем в спецификации пакета.

Пример

Для удаления заголовка otag из пакета можно задать

```
delete_header(egr_pkt.group1.otag);
```

Это работает с экземпляром пакета, удаляя из него otag без влияния на дерево синтаксического анализа.

Для удаления группы заголовков можно использовать

```
delete_header(egr_pkt.group1);
```

Это работает с экземпляром пакета, удаляя из него группу заголовков group1 без влияния на дерево анализа.

4.10.3. Перезапись заголовка

Для некоторых протоколов при обработке пакета требуется изменять некоторые поля заголовка.

Таблица 12. Конструкция replace_header_field.

Конструкция	Аргументы, опции	Описание
replace_header_field		Заменяет поле заголовка полем из лины или другим полем заголовка.
	dest_field	Имя изменяемого поля заголовка.
	src_field	Имя поля, используемого в качестве источника при замене (bus.field или header.field).

Пример

Для изменения поля dscp можно использовать

```
replace_header_field(egr_pkt.ipv4.dscp, obj_bus.new_dscp);
```

4.10.4. Создание контрольной суммы

Конструкция create_checksum может использоваться только в функциях и поддерживает расчёт контрольных сумм TCP и UDP.

Таблица 13. Конструкция create_checksum.

Конструкция	Аргументы, опции	Описание
create_checksum		Создаёт контрольную сумму.
	checksum_field	Задаёт имя поля контрольной суммы в пакете (<packet.field>).
	<packet_field_list>	Упорядоченный список полей для расчёта контрольной суммы. <packet._PAYLOAD> считается флагом включения данных в контрольную сумму.

```
create_checksum(egress_pkt.group2.ipv4.hdr_checksum,
        {egress_pkt.group2.ipv4.version, egress_pkt.group2.ipv4.hdr_len,
        egress_pkt.group2.ipv4.tos, egress_pkt.group2.ipv4.v4_length,
        egress_pkt.group2.ipv4.id, egress_pkt.group2.ipv4.flags,
        egress_pkt.group2.ipv4.frag_offset, egress_pkt.group2.ipv4.ttl,
        egress_pkt.group2.ipv4.protocol, egress_pkt.group2.ipv4.sa,
        egress_pkt.group2.ipv4.da});

create_checksum(egress_pkt.fwd_l3_l4_hdr.udp.checksum,
        {egress_pkt.fwd_l3_l4_hdr.ipv4.sa,
        egress_pkt.fwd_l3_l4_hdr.ipv4.da,
```

```
        editor_dummy_bus.zero_byte,
        egress_pkt.fwd_l3_l4_hdr.ipv4.protocol,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp.src_port,
        egress_pkt.fwd_l3_l4_hdr.udp.dst_port,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp._PAYLOAD});
```

4.10.5. Обновление размера пакета

Конструкция update_packet_length может применяться только в функциях.

Таблица 14. Конструкция update_packet_length.

Конструкция	***Аргументы, опции***	***Описание***
update_packet_length		Обновляет размер пакета.
	packet_length_field	Задаёт имя поля размера в пакете (<packet.field>).
	update_type	Задаёт тип обновления размера пакета: - 0 указывает использование лишь размера данных (payload) пакета; - 1 указывает использование данных и заголовка.
	truncate_mode	Указывает выполнение отсечки пакета: - 0 - пакет не усекается; - 1 - пакет усекается.

```
update_packet_length(egress_pkt.group2.ipv4.v4_length, 1);
```

Пример

Использование create_checksum и update_packet_length

```
function do_checksum_update() {
        create_checksum(egress_pkt.group2.ipv4.hdr_checksum,
                {egress_pkt.group2.ipv4.version, egress_pkt.group2.ipv4.hdr_len,
                egress_pkt.group2.ipv4.tos, egress_pkt.group2.ipv4.v4_length,
                egress_pkt.group2.ipv4.id, egress_pkt.group2.ipv4.flags,
                egress_pkt.group2.ipv4.frag_offset, egress_pkt.group2.ipv4.ttl,
                egress_pkt.group2.ipv4.protocol, egress_pkt.group2.ipv4.sa,
                egress_pkt.group2.ipv4.da});
        create_checksum(egress_pkt.group4.ipv4.hdr_checksum,
                {egress_pkt.group4.ipv4.version, egress_pkt.group4.ipv4.hdr_len,
                egress_pkt.group4.ipv4.tos, egress_pkt.group4.ipv4.v4_length,
                egress_pkt.group4.ipv4.id, egress_pkt.group4.ipv4.flags,
                egress_pkt.group4.ipv4.frag_offset, egress_pkt.group4.ipv4.ttl,
                egress_pkt.group4.ipv4.protocol, egress_pkt.group4.ipv4.sa,
                egress_pkt.group4.ipv4.da});
}

function do_packet_length_update() {
        update_packet_length(egress_pkt.group2.ipv4.v4_length, 1);
        update_packet_length(egress_pkt.group4.ipv4.v4_length, 1);
}

program app {
        ...
        do_packet_length_update();
        do_checksum_update();
        ...
}
```

5. Конструкции для целевой платформы

В конвейере обработки целевой платформы могут быть базовые утилиты, ускорители, настраиваемые компоненты, обеспечивающие эффективную реализацию некоторых сетевых функций. NPL поддерживает конструкции для определения и вызова таких компонентов вместе с остальными логическими функциями. Производитель платформы определяет, а разработчик программы NPL может вызывать из своего приложения:

1. внешние функции платформы;
2. специальные функции;
3. динамические таблицы.

Внешними функциями целевой платформы являются базовые функции целевой архитектуры. Внешние функции можно неоднократно вызывать из приложения вместе с другими конструкциями NPL. Например, внешняя функция отбрасывания пакетов может вызываться как часть поиска logical_table.

Конструкции специальных функций используются для указания ускорителей или блока IP[1] целевой платформы и режимов их использования. Для специальных функций нужны особые соединения в терминах входов и выходов. Внутреннее устройство специальных функций на задаётся в NPL, это остаётся за производителем, программы NPL просто вызывают функции.

Конструкции логических таблиц служат для задания таблиц целевой платформы, применяемых в процессе работы. Производитель платформы задаёт базовую структуру динамических таблиц, а разработчик NPL - набор логических сигналов, позволяющих SDK целевой платформы создавать логические таблицы в процессе работы. После

[1] *Intellectual Property - закрытый блок, защищенный авторским правом. Прим. перев.*

преобразования NPL в язык модели, такой как C++, поведение специальных и внешних функций определяется целевой платформой.

5.1. Внешние функции целевой платформы

Каждое сетевое устройство имеет набор фундаментальных функций, вызываемых многократно. Например, отбрасывание пакета, копирование в CPU или другой порт для трассировки или подсчёта пакетов. Эти базовые функции могут быть связаны с logical_table, функцией или иной конструкцией NPL. Например, отбрасывание пакета является частью поиска в логической таблице, а отображение пакета (mirroring) - частью функции обработки.

NPL позволяет производителям платформ задавать такие функции как внешние. Производитель задаёт шаблон внешней функции с информацией, требуемой для эффективного использования оборудования.

5.1.1. Определение внешней функции

Определяемый производителем шаблон внешней функции аналогичен применяемым в других языках.

Таблица 15. Конструкция extern.

Конструкция	***Аргументы, опции***	***Описание***
extern <имя функции>		Заданная производителем внешняя функция.
	direction	in или out.
	Имя поля	Задаёт поля с размером и типом.

Пример

Определение внешней функции для отбрасывания пакетов

```
extern packet_drop(in bit[1] trigger, in const value, in const drop_code);
```

Здесь trigger указывает то или иное поле шины, которое может вызывать отбрасывание пакета. Остальные свойства связаны с процедурой отбрасывания.

5.1.2. Применение внешних функций

Разработчик программ NPL может вызывать внешние функции из приложения, помещая вызовы в logical_table или function.

Пример

```
logical_table packet_integrity {
....
        fields {
                bit copy_to_cpu;
                bit pkt_integrity_drop;
        }
......
        fields_assign() {
                ....
                packet_drop(pkt_integrity_drop, drop_reason.PKT_INTEGRITY_CHECK_FAILED, 2);
        }
}
```

5.2. Конструкция для специальных функций

5.2.1. Определение специальной функции

Конструкция special_function служит для задания интерфейса с IP-блоком целевой архитектуры и внутренняя функциональность блока IP не задаётся в NPL. Определение интерфейса с блоком IP должно предоставляться производителем на основе использования шаблонов методов. Разработчик программы NPL вызывает блок IP из программы, используя заданные шаблоны и подходящие аргументы. Конструкция special_function является расширяемой и производитель может добавлять методы.

Таблица 16. Конструкция special_function.

Конструкция	***Аргументы, опции***	***Описание***
special_function		Заданная производителем внешняя функция.
	special_function_name	Задаёт имя IP-блока.
	<generic method>([in/out] [const] или [auto_enum] или [str] или [bit/varbit] [size] или [list] [name])	Прототип метода IP-блока, задающий направление и типа аргументов. Предоставляется производителем платформы. Компилятор для платформы должен обеспечивать обработку определений и вызовов. Поддерживается множество методов. Тип varbit указывает маскируемый аргумент.

5.2.1.1. Пример задания special_function для целевой платформы

Производитель платформы может применять разные методы для задания интерфейса с IP-блоком.

Пример

```
special_function flex_qos_phb {
        usage_mode_create(in const index,
                in bit[10]
                qos_base,
                in varbit[6] qos_attr,
                out bit[4]
                int_pri);
        usage_mode_select(in bit[6] eindex);
}
```

5.2.2. Использование специальных функций

Программист NPL связывает блоки IP с логической функциональностью в программе NPL, используя:

- методы из библиотеки special_function целевой платформы;
- встроенный в NPL метод execute().

5.2.2.1. Методы специальных функций

Разработчик программ NPL использует методы special_function, предоставляемые целевой архитектурой, для задания соединений с блоками IP. Размещение вызовов этих блоков в коде NPL не отражает последовательность вызовов. Синтаксис вызова показан ниже. Тип и порядок аргументов должны совпадать с указанными в шаблоне. Аргументы метода special_function method передаются по ссылкам.

```
<special_function_name>.<method_name>(<arguments>)
```

5.2.2.2. execute()

Встроенный метод execute() не задаётся в конструкции special_function. Этот метод активирует блок IP и обеспечивает его относительную позицию в логической функциональности. Метод не использует аргументов.

Таблица 17. Конструкция execute().

Конструкция	*Аргументы, опции*	*Описание*
execute()		Встроенный метод активирования блока IP.

5.2.2.3. Пример использования special_function для целевой платформы

Программист NPL может получить доступ к IP-блоку целевой платформы, используя прототипы метода special_function.

Пример

```
program app {
        ...
        flex_qos_phb.usage_mode_create(flex_qos_entry.QOS_MPLS_EXP_L3_TUNNEL_ECN,
                ing_obj_bus.mpls_exp_mapping_ptr[9:0],
                ing_cmd_bus.mpls_effective_exp_for_phb,
                ing_cmd_bus.int_pri);
        flex_qos_phb.usage_mode_create(flex_qos_entry.QOS_MPLS_EXP_L2_TUNNEL_ECN,
                ing_obj_bus.mpls_exp_mapping_ptr[9:0],
                ing_cmd_bus.mpls_effective_exp_for_phb1,
                ing_cmd_bus.int_pri1);
        ...
        flex_qos_phb.usage_mode_select(phb_select_lts_tcam_key.entry_index);
        flex_qos_phb.execute();
        ...
}
```

5.3. Конструкции для динамических таблиц

5.3.1. dynamic_table

Конструкция dynamic_table служит для задания логических таблиц в процессе работы и обслуживается целевой платформой. Разработчик программы NPL задаёт набор логических сигналов, позволяющих SDK создавать таблицы.

Определение dynamic_table предоставляет производитель платформы и оно не имеет явных входных или выходных соединений, лишь указывая SDK, как использовать динамические таблицы. Таблица может иметь множество методов для поддержки разных целей. Размер dynamic_table определяется равным 1.

Таблица 18. Конструкция dynamic_table.

Конструкция	*Аргументы, опции*	*Описание*
dynamic_table		Задаёт динамическую таблицу.
	dynamic_table_name	Имя блока динамической таблицы
	<generic method>([in/out] [list] [name])	Прототип метода, указывающий направление и список аргументов. Предоставляется производителем платформы. Компилятор для платформы должен обеспечивать обработку определений и вызовов. Поддерживается множество методов.

5.3.1.1. Пример определения dynamic_table для целевой платформы

Производитель целевой платформы может применять приведённые ниже методы для определения интерфейса динамической таблицы.

Пример

```
dynamic_table ing_fp {
        presel_template(in list presel_menu);
        rule_template(in list rule_menu);
        action_template(out list action_menu);
}
```

5.3.2. Использование динамических таблиц

Программист NPL использует шаблоны методов динамической таблицы для сопоставления с ней логических сигналов.

5.3.2.1. <dynamic_table_name>.<method_name>(<argument_list>)

Разработчик NPL применяет методы dynamic_table, предоставленные целевой архитектурой, для связывания логических сигналов с блоками динамической таблицы. Местоположение вызовов методов в NPL не отражает реальный порядок вызовов. Формат вызова шаблона метода имеет вид

`<dynamic_table_name>.<method_name>(<argument_list>)`
Аргументы метода динамической таблицы передаются ссылками.

5.3.2.2. lookup()
Встроенный метод lookup() не задаётся в конструкции dynamic_table и служит для задания места вызова dynamic_table.

Таблица 19. Конструкция lookup().

Конструкция	*Аргументы, опции*	*Описание*
lookup()		Встроенный метод для вызова блока динамической таблицы.

5.3.2.3. Образец применения dynamic_table для целевой платформы
Разработчик NPL может получить доступ к блоку dynamic_table целевой платформы, используя прототипы методов dynamic_table.

Пример

```
program {
        ...
        ing_fp.presel_template({
                ing_cmd_bus.l2_iif_opaque_ctrl_id,
                ing_cmd_bus.l3_iif_opaque_ctrl_id,
                ...
        });
        ing_fp.rule_template({
                pkt_fwd_field_bus.macda,
                pkt_fwd_field_bus.macsa,
                ...
        });
        ing_fp.action_template({
                ifp_scratch_bus.ifp_drop_action,
                ifp_scratch_bus.ifp_drop_code,
                ...
        });
ing_fp.lookup(); // поиск служит конструктором для dynamic_table
}
```

6. Конструкции сравнения силы
В парадигме NPL может одновременно выполняться поиск в нескольких таблицах. Когда несколько таблиц назначает (assign) один и тот же объект, нужен механизм выбора между ними. В NPL используется механизм выбора на основе «силы» (strength), которая задаётся численным значением.

В NPL имеются конструкции для связывания значений силы с результатами поиска. При каждом поиске создаётся профиль силы для результата. Значение силы может быть статическим (для таблицы) или динамическим (для записи).

Для выбора на основе силы в NPL используется несколько конструкций:

- сила записей логических таблиц;
- таблица fields_assign() для указания необходимости сравнения силы;
- функция сравнения силы.

6.1. Создание логической таблицы силы
Конструкция strength объявляет прототип для записей логической таблицы силы, которая может содержать одно или несколько полей strength. Для создания таблицы применяются конструкции struct, указывающие поля, нужные для сравнения силы. Эта структура может включать лишь поля типа bit (но не вложенные struct). Экземпляры таблиц создаются с помощью конструкции strength, имя которой должно быть уникальным в глобальном масштабе. Записи таблицы для поиска при сравнении силы задаются конструкциями strength_resolve.

Таблица 20. Конструкция strength.

Конструкция	*Аргументы, опции*	*Описание*
strength		Создаёт экземпляр таблицы силы.
	Имя struct	Имя структуры таблицы силы, имена полей которой представляют элементы записи таблицы силы (тип struct).
	Имя таблицы	Имя таблицы силы в форме строки (string).

6.2. Присоединение таблицы силы
В логической таблице fields_assign() следует применять конструкцию use_strength вместо оператора присваивания для задания сравнения силы. Применение конструкции use_strength задаёт индекс таблицы силы, служащий для выбора записи, которая будет определять значение силы для результата поиска в таблице.

Таблица 21. Конструкция use_strength.

Конструкция	*Аргументы, опции*	*Описание*
use_strength		Привязывает таблицу силы к логической таблице.
	strength table name	Имя таблицы силы, объявленное со структурой strength.

index	Индекс таблицы силы, которая будет применяться для сравнения результатов поиска в исходной логической таблице. Размер таблицы профиля силы определяется числом битов этого аргумента. В случае постоянных индексов размером будет число битов, требуемых для наибольшего значения. При размере B таблица будет иметь размер 2B. Индекс может быть: - константой, определяемой при компиляции; - полем данной логической таблицы (динамическая сила); - полем шины.

6.3. Конструкция strength_resolve

Конструкция strength_resolve задаёт объект шины, который назначается с помощью сравнения силы. Она также задаёт (strength_list) значение силы для каждого сравниваемого поиска logical_table. Соответствующие записи (source_field_list) связывают значения силы с результатами поиска в таблице, которые нужно сравнить (индексы use_strength). Таблица с наибольшим значением силы будет предоставлять значение объекта шины.

Таблица 22. Конструкция strength_resolve.

Конструкция	***Аргументы, опции***	***Описание***
strength_resolve		Задаёт способ назначения объекта выбранного из разных источников.
	destination bus.field	Поле шины bus.field, куда передаётся объект после сравнения силы.
	destination bus.field strength	Сила, связанная с целевым bus.field (bus.field или NULL).
	<strength_entry_0>	Список свойств, описывающих первый элемент силы (см. strength_entry).
	<strength_entry_1>	Список свойств, описывающих второй элемент силы (см. strength_entry).
	...	
	<strength_entry_n>	Список свойств, описывающих n-ый элемент силы (см. strength_entry).
strength_entry		

```
{table_lookup, user_defined_view_type, strength, source_field}
```

table_lookup	Указывает поиск в таблице, где source_field имеет значение table.field (table._LOOKUP0 или table._LOOKUP1).
user_defined_view_type	Задаёт тип представления поля, заданный пользователем. Используется то же значение auto_enum, которое задано в блоке fields_assign(). Возможны значения auto_enum и NULL.
strength	Сила, связанная с результатом logical_table, соответствующим сравниваемому объекту (поле силы table.field).
source_field	Исходное поле, которое может быть назначено целевому полю (logical_table.field).

Пример

Указание статической силы для объекта, полученного из 2 таблиц

Пусть псевдокод для сравнения силы имеет вид

```
if (obj_strength_profile[1].obj_k_str > obj_strength_profile[2].obj_k_str)
        cmd_bus.obj_k = Table_A.obj_k;
else
        cmd_bus.obj_k = Table_B.obj_k;
```

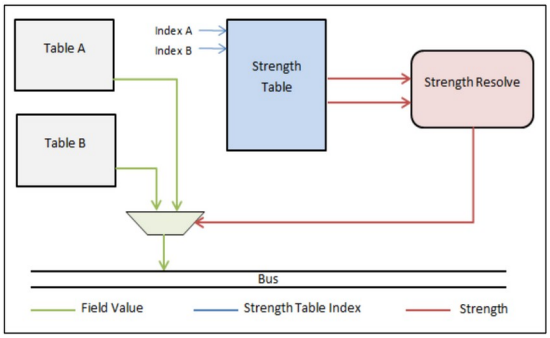


Рисунок 3. Сила при использовании таблиц со статическим индексированием.

Конструкция будет иметь вид

```
struct cmd_bus_t {
        fields {
                bit[8]  obj_k;
        }
}

struct obj_strength_t {
        fields {
                bit[4]  obj_k_str;
        }

bus cmd_bus_t   cmd_bus;
strength obj_strength_t obj_strength_profile; // таблицы профилей силы

logical_table Table_A {
        ...
        fields {
                bit[8]  obj_k;
        }
```

```
        fields_assign() {
                use_strength(obj_strength_profile, 1);  // ссылка на запись таблицы профилей силы
        }
}

logical_table Table_B {
        ...
        fields {
                bit[8]  obj_k;
        }
        fields_assign() {
                use_strength(obj_strength_profile, 2); // ссылка на запись таблицы профилей силы
        }
}

program app {
        ...
        strength_resolve(cmd_bus.obj_k, NULL,
                { Table_A._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_A.obj_k},
                { Table_B._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_B.obj_k});
        ...
}
```

Указание динамической силы для объекта, полученного из 2 таблиц

Пусть псевдокод для сравнения силы имеет вид

```
if (obj_strength_profile[index_A].obj_k_str > obj_strength_profile[index_B].obj_k_str)
        cmd_bus.obj_k = Table_A.obj_k;
else
        cmd_bus.obj_k = Table_B.obj_k;
```

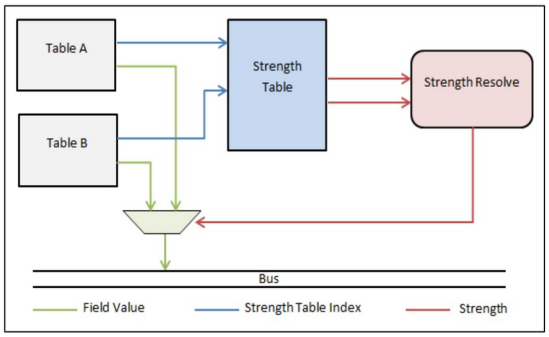


Рисунок 4. Сила при использовании таблиц со динамическим индексированием.

Конструкция будет иметь вид

```
struct cmd_bus_t {
        fields {
                bit[8]  obj_k;
        }
}

struct obj_strength_t {
        fields {
                bit[4]  obj_k_str;
        }
}

bus cmd_bus_t cmd_bus;
strength obj_strength_t obj_strength_profile;

logical_table Table_A {
        ...
        fields {
                bit[8]  obj_k;
                bit[5]  strength_index;
        }
        fields_assign() {
                use_strength(obj_strength_profile, strength_index);
        }
}

logical_table Table_B {
        ...
        fields {
                bit[8]  obj_k;
                bit[5]  strength_index;
        }
        fields_assign() {
                use_strength(obj_strength_profile, strength_index);
        }
}
program app {
        ...
        strength_resolve(cmd_bus.obj_k, NULL,
```

```
                { Table_A._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_A.obj_k},
                { Table_B._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_B.obj_k});
        ...
}
```

Указание силы для объекта, полученного из таблицы и шины

Пусть псевдокод для сравнения силы имеет вид

```
if (obj_strength_profile[a_strength_index].obj_k_str > cmd_bus.obj_k_str)
        cmd_bus.obj_k = Table_A.obj_k;
        else
cmd_bus.obj_k = cmd_bus.obj_k;
```

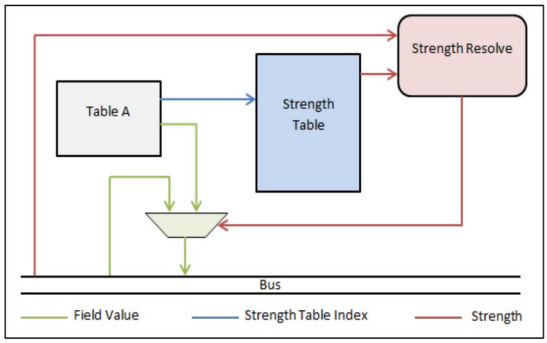


Рисунок 5. Сила при использовании таблицы и шины.

Конструкция будет иметь вид

```
struct cmd_bus_t {
        fields {
                bit[8]  obj_k;
                bit[4]  obj_k_str;
        }
}
struct obj_strength_t {
        fields {
        bit[4]  obj_k_str;
        }
}

bus cmd_bus_t cmd_bus;
strength obj_strength_t obj_strength_profile;

logical_table Table_A {
        ...
        fields {
                bit[8]  obj_k;
                bit[3]  a_strength_index;
        }
        fields_assign() {
                use_strength(obj_strength_profile, a_strength_index);
        }
}

program () {
        strength_resolve(cmd_bus.obj_k, cmd_bus.obj_k_str,
                {Table_A._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_A.obj_k});
}
```

Указание силы при использовании двух таблиц с множественным поиском

```
struct obj_bus_t {
        fields {
                bit[12] dst_vlan;
                bit[12] src_vlan;
                bit[11] dst_vfi;
                bit[11] src_vfi;
        }
}

struct cmd_bus_t {
        fields {
                bit     dst_discard;
                bit     src_discard;
        }
}

struct UAT_strength_profile_t {
        fields {
                bit[4] obj_src_discard_str;
                bit[4] obj_K_str;
        }
}

strength UAT_strength_profile_t UAT_strength_profile;
bus obj_bus_t obj_bus;
```

```
bus cmd_bus_t cmd_bus;

// множество поисков
logical_table Table_A {
        ...
        field {
                bit[12] vlan;
                bit     discard; // поле сравнения силы
        }
        fields_assign() {
                if (_LOOKUP0)
                        obj_bus.dst_vlan = vlan;
                if (_LOOKUP1)
                        obj_bus.src_vlan = vlan;
                use_strength(UAT_strength_profile, 10);
        }
}

// один поиск, не нужно добавлять _LOOKUP0 в fields_assign().
logical_table Table_C {
        ...
        field {
                bit[11] vfi;
                bit     discard; // поле сравнения силы
        }
        fields_assign() {
                obj_bus.dst_vfi = vfi;
        }
}

// То же, что Table_A, но с заданной ниже логикой силы
logical_table Table_B {
        ...
        field {
                bit[11] vfi;
                bit     discard; // поле сравнения силы
        }
        fields_assign() {
                if (_LOOKUP0) {
                        obj_bus.dst_vfi = vfi;
                }
                if (_LOOKUP1) {
                        obj_bus.src_vfi = vfi;
                }
                use_strength(UAT_strength_profile, 20);
        }
}
program () {
        ...
        // два поиска
        Table_A.lookup(0);
        Table_A.lookup(1);
        // один поиск
        Table_C.lookup(0);

        // два поиска
        Table_B.lookup(0);
        Table_B.lookup(1);
        strength_resolve(cmd_bus.src_discard, NULL,
          {Table_A._LOOKUP1, NULL, UAT_strength_profile.obj_src_discard_str, Table_A.discard},
          {Table_B._LOOKUP1, NULL, UAT_strength_profile.obj_src_discard_str, Table_B.discard});
        strength_resolve(cmd_bus.dst_discard, NULL,
          {Table_A._LOOKUP0, NULL, UAT_strength_profile.obj_K_str, Table_A.discard},
          {Table_B._LOOKUP0, NULL, UAT_strength_profile.obj_K_str, Table_B.discard});
        ...
}
```

6.4. Сравнение силы с помощью функции

Конструкция function может также применяться для сравнения силы при последовательном обращении к таблицам. При компиляции может быть выбрана иная схема отображения.

7. Базовые конструкции

7.1. Атрибуты NPL

Атрибуты NPL служат для передачи намерений программы NPL в файлы yaml, которые будут доступны для извлечения соответствующей информации. Атрибуты помещаются в «скобки» <! и !>. Компилятор не пытается проверять содержимое атрибутов NPL.

7.1.1. Позиционные атрибуты

Позиционные атрибуты служат для прикрепления документации к коду NPL с целью описания различных элементов. Это отличается от комментариев (// и /* */), поддерживаемых в NPL, хотя те и другие не влияют на код и компиляцию.

Атрибуты NPL поддерживаются для перечисленных ниже элементов:

- logical_table - ключи, поля;
- logical_register - поля;
- шина (struct) - поля, наложения;
- заголовок пакета (struct) - поля;
- special_function и dynamic_table.

Компиляторы могут распространять атрибуты NPL в выходные файлы для целевой платформы. Например, целевой компилятор может распространить документацию logical_table в выходной файл Regsfile или Map.

Таблица 23. Позиционные атрибуты.

Дескриптор	***Назначение***	***Описание***
REGSFILE, DESC: <desc>	logical_regsfile.yml header.yml	Применимы к logical_table, logical_table.field, logical_table.key, logical_register, logical_register.field, struct, (bus/header) struct.field. (bus/header) Атрибут <desc> применяется как TABLE <tbl> DESC, TABLE <tbl.fld> DESC, REGISTER <reg> DESC, REGISTER <reg.fld> DESC, FORMAT <struct> DESC, FORMAT <struct.fld> DESC, HEADER <struct> DESC, HEADER <struct.fld> DESC.
REGSFILE, ENCODING: <enum>	logical_regsfile.yml	Применимы к logical_table.field. Атрибут <enum> применяется как поле таблицы, ENCODINGS
REGSFILE, ENCODING: DESC: enum.field = <desc>	logical_regsfile.yml	Применимы к logical_table.field, enum.field. Атрибут <desc> применяется как поле logical_table, поле ENCODINGS, DESC.
REGSFILE, FIELD_NAME: <field>	logical_sftblfile.yml	Применимы к dynamic_table.arg. Атрибут <field> применяется как dyamic_table.field

Пример (REGSFILE, DESC: <desc>)

Код NPL

```
<!(REGSFILE, DESC: "This table is looked up using Layer 2 incoming interface packet was received on. This
table provides incoming Layer 2 interface attributes for the packet.") !>
logical_table ing_l2_iif_table {
        ...
        fields {
                <!(REGSFILE, DESC: "If set, IPV6 Tunnel is enabled on this interface.") !>
                bit ipv6_tunnel_enable;
        }
}
```

Представление Regsfile

```
TABLE = {
        ing_l2_iif_table:
        DESC: |-
                This table is looked up using Layer 2 incoming interface packet was received on.
                This table provides incoming Layer 2 interface attributes for the packet.
        FIELDS:
                ipv6_tunnel_enable:
                        DESC: |-
                                If set, IPV6 Tunnel is enabled on this interface.
                        MAXBIT: 13
                        MINBIT: 13
                        ORDER: 4
                        TAG: data
                        WIDTH: 1
```

Пример (REGSFILE, ENCODING: <enum>)/(REGSFILE, ENCODING: DESC: enum.field = <desc>)

Код NPL

```
enum pvlan_port {
        PVLAN_PROMISCUOUS_PORT = 0,
        PVLAN_COMMUNITY_PORT = 1,
        PVLAN_ISOLATED_PORT = 2
}
logical_table ing_vfi_table {
        ...
        fields {
                <!REGSFILE, ENCODING: pvlan_port !> !
                <!REGSFILE, ENCODING: DESC: pvlan_port.PVLAN_PROMISCUOUS_PORT = Pvlan Promiscuous
port !>
                bit[FIELD_2_WD] src_pvlan_port_type;
        }
}
```

Представление Regsfile

```
ing_vfi_table:
        FIELDS:
                src_pvlan_port_type:
                        ENCODINGS:
                                pvlan_port__PVLAN_PROMISCUOUS_PORT:
                                        DESC: |-
```

```
                                        Pvlan Promiscuous port
                                VALUE: 0
                        pvlan_port__PVLAN_COMMUNITY_PORT:
                                DESC: ''
                                VALUE: 1
                        pvlan_port.PVLAN_ISOLATED_PORT
                                DESC: ''
                                VALUE: 2
```

Пример (REGSFILE, FIELD_NAME: <field>)

Код NPL

```
dynamic_table egr_flex_ctr {
        presel_template (in list presel_menu);
        object_template (in list object_menu);
}
        egr_flex_ctr.presel_template(
        {
                <!REGSFILE, FIELD_NAME: "mirror_pkt_ctrl_0" !>
                egr_cmd_bus.mirror_pkt_ctrl
        });
```

Представление Regsfile

```
dt_egr_flex_ctr_presel_template:
        FIELDS:
                mirror_pkt_ctrl_0:
                        BUS_SELECT_WIDTH: 4
                        DESC: |-
                                Input - egr_cmd_bus_mirror_pkt_ctrl
                        MAXBIT: 65
                        MINBIT: 2
                        ORDER: 2
                        TAG: bus_select
                        WIDTH: 64
```

7.1.2. Непозиционные атрибуты

Непозиционные атрибуты позволяют разработчику NPL полностью передать свои намерения. Все такие атрибуты собираются в выходном файле IFILE (Intent File) формата yaml. Другие атрибуты в IFILE не включаются. В файле атрибуты группируются по указанной функциональности, которую они поддерживают. Программист NPL отвечает за указание формата содержимого непозиционных атрибутов, а потребитель таких атрибутов - за их анализ и преобразование в желаемый формат, а также проверку атрибутов. Потребитель IFILE должен создать утилиту для приёма выхода компилятора и преобразования его в желаемый формат с проверкой.

Непозиционные атрибуты NPL передаются в IFILE на основе описаний.

Таблица 24. Непозиционные атрибуты.

Дескриптор	***Назначение***	***Описание***
(IFILE, <blk>: <code>)	ifile.yml	Поле <code> добавляется к блоку <blk> в файле IFILE.

7.1.2.1. Инициализация

Программист NPL может указать использование логических таблиц, таблиц силы и других элементов.

Пример

Код NPL

```
// назначение логической таблицы NPL
<! IFILE, INIT: "lt ing_l3_next_hop_1_table nhop_index_1=0 dvp=0x0 l3_oif_1=0x0" !>

// назначение физического ресурса по потребности
<! IFILE, INIT: "pt IFTA150_SBR_PROFILE_TABLE_0_INDEX=10 strength=0x5" !>

// назначение символьного элемента (в будущем)
<! IFILE, INIT: "lt _SYMBOL=TIMESTAMP _INDEX=10 data=0x5" !>

// использование перечисляемого NPL
<! IFILE, INIT: "pt EPOST_FMT_AUX_BOTP_IN_DATA mtu_drop=NPL_EGR_MTU_DROP_ENUM" !>
```

Выходной IFILE

```
INIT:
        0: |-
                lt ing_l3_next_hop_1_table nhop_index_1=0 dvp=0x0 l3_oif_1=0x0
        1: |-
                pt IFTA150_SBR_PROFILE_TABLE_0_INDEX=10 strength=0x5
        2: |-
                lt _SYMBOL=TIMESTAMP _INDEX=10 data=0x5
        3: |-
                pt EPOST_FMT_AUX_BOTP_IN_DATA mtu_drop=NPL_EGR_MTU_DROP_ENUM
```

7.1.2.2. Relational

Программист NPL может указать дополнительную функциональность, относящуюся к символам или вызовам NPL.

Пример

Код NPL

```
<! IFILE, REL: "ecmp_level0:npl_ecmp_level0_member_table" !>
```

Выходной IFILE

```
REL:
        0: |-
                ecmp_level0:npl_ecmp_level0_member_table
```

7.2. Конструкции препроцессора

7.2.1. #include

Директива #include служит для включения исходного кода NPL в другую программу NPL.

```
#include "bus.npl"                    // файл из того же каталога
#include "../lib/header.npl"          // файл из другого каталога
```

7.2.2. #if - #endif

NPL поддерживает условную компиляцию в стиле C/C++.

```
#ifdef XYZ
```

7.2.3. #define

NPL поддерживает макросы в стиле C/C++ для переменных, но без параметризации. Определения задаются заглавными буквами.

```
#define CPU_PORT                5
#define MAC_ADDR_BCAST  0xffffffffffff
```

7.3. Комментарии

NPL поддерживает 2 варианта комментариев:

- многострочные с использованием символов /* и */. Такие комментарии можно включать в строку (in-line);
- однострочные от символов // до конца строки.

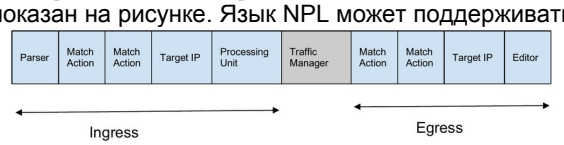
7.4. print

NPL использует конструкцию print для вывода значений переменных в программе. Команда print транслируется в модель Behavioral C. Это не имеет значения с точки зрения компиляции. Вызов print в модели C выполняется в том же порядке, что и в программе NPL. Печатаются только поля (не struct). Команда print работает аналогично C printf.

```
print("Value of the SVP is %d, VFI is %d\n", obj_bus.svp, obj_bus.vfi);
```

8. Приложение A. Пример конвейера

Пример конвейера в коммутаторе показан на рисунке. Язык NPL может поддерживать разную архитектуру.



Parser задаёт и изменяет поведение аппаратного блока синтаксического анализа (HW Parser Block).

Match Action - логические таблицы и регистры NPL, задающие и изменяющие поведение блоков «сопоставление-действие» (Match Action Block).

Target IP - блок IP, относящийся к производителю платформы и представленный конструкцией special_function.

Processing Unit - базовый элемент обработки на целевой платформе, поведение которого задают и изменяют функции NPL и сравнение силы.

Editor - редактор NPL, определяющий поведение блока редактирования.

Целевая платформа может использовать несколько экземпляров этих блоков в произвольном порядке.

9. Приложение B. Рекомендации по использованию

9.1. struct в заголовках

Объекты верхнего уровня (например, logical_table, logical_register, enum) не могут быть частью struct. NPL поддерживает вложенные структуры с учётом приведённых ниже ограничений.

Таблица 25. Использование struct.

Применение	***bit/bit[n]***	***overlay***	***struct***	***Вложенные struct***
Тип header	+	-	-	-
Тип header_group	-	-	+	-
Пакет	-	-	+	-
Шина	+	+	+	+

Таблица 26. Ссылки на struct в пакетах.

Действительная конструкция	***Возможно***	***Описание ссылки***
packet.struct.struct.field	+	packet.group.header.field
packet.struct.struct	+	packet.group.header
packet.struct.field	-	Нет группы/заголовка
packet.field	-	Пакет должен иметь группу/заголовок
Нет пакета	-	

9.2. Функции

Функции с аргументами не поддерживаются. Функции должны работать с логическими шинами и данными пакетов.

9.3. Правила наложения

Наложения могут применяться во многих конструкциях и должны следовать приведённым ниже правилам.

1. Поля наложения могут быть заданы для базовых полей типа bit и bit[n].
2. Поля наложения могут быть заданы для базовых полей типа struct (т. е., полей, заданных как struct в шине). Однако для struct не разрешено частичное наложение.
3. Наложения могут перекрываться.
4. Не допускается задание наложений для других полей наложения.
5. Наложенные поля не могут иметь тип struct.

9.4. «Нарезка» битовых массивов

NPL поддерживает «нарезку» (диапазон) битовых массивов (bit-array) в назначениях, уравнениях, специальных функциях. Диапазоны битов можно указывать в обеих частях (lvalue и rvalue) уравнений.

Пример

```
a = b[7:4];
if (b[5:3])...
obj_bus.a[5:4] = ing_pkt.ipv4.ecn;      // в основном функции и действия
a = b[0:0];                             // доступ к одному биту с указанием диапазона 1 бит
```

Варианты применения

```
1. if (a[5:3])              // корректно
2. a[5:3] = b[5:3];         // корректно
```

9.5. Правила конкатенации

- При назначении конкатенация разрешена лишь в правой стороне уравнений. Обычно она применяется для полей varbit.

  ```
  a = b<>c;
  ```

- Конкатенация не разрешена в левой части уравнения.

  ```
  b<>c = a[5:0];
  ```

- Конкатенация разрешена для однотипных полей (например, struct<>struct).

  ```
  two_mpls_hdrs = mpls_hdr_0<>mpls_hdr_1;
  ```

- Конкатенация разрешена для разнотипных полей (например, struct<>bit).

  ```
  new_mpls_hdr = mpls_hdr_0<>c[3:0];
  ```

- Конкатенация разрешена для частей полей (например, da[5:0]<>sa[5:3]).

  ```
  a[5:0] = b[3:2]<>c[3:0];
  ```

- При сравнении со значением регистра поддерживается лишь оператор ==.

10. Приложение C. Зарезервированные слова NPL

fields_assign	add_header	auto_enum	bit
bus	create_checksum	default	delete_header
define	dynamic_table	else	enum
extract_fields	fields	function	hash
header_length_exp	if	index	keys
latest	logical_register	logical_table	mask
maxsize	minsize	next_node	NPL_PRAGMA
overlays	packet	keys_construct	parse_break
parse_continue	parse_begin	end_node	parser_node
print	program	replace_header_field	root_node
special_function	strength	strength_resolve	struct
switch	table_type	tcam	update_packet_length
use_strength	varbit	_HIT_INDEXx	_LOOKUPx
_PRESENT	_VALID	extern	true
false			

11. Приложение D. Грамматика NPL

В этом приложении описана грамматика NPL с использованием нотации yacc.

Лексер маркирует идентификаторы (ID) для заданных пользователем имён регулярным выражением [A-Za-z_][\w_]*. Десятичные константы должны быть натуральными числами. Шестнадцатеричные константы задаются в обычной форме (например, 0x0f, 0X0f, 0x0F). Константы размера подобны шестнадцатеричным константам с размерами в стиле printf (например, 8x00). Строковые константы должны заключаться в двойные кавычки и не содержать символов новой строки (например, “foo.bar”).

Идентификаторы

```
primary_types:  constant
                      | identifier

                      |STR_CONST      /* r'\"([^\\\n]|(\\.))*?\"' */
```

```
constant :              |DEC_CONST      /* ([0-9][0-9]*) */
                        |HEX_CONST      /* (0[xX][0-9A-Fa-f]+) */

identifier : ID

dir :   IN
        | OUT
```

Объявления

```
npl_node :      npl_declaration_specifier
                | empty

npl_declaration_specifier :     npl_declaration
                                | npl_declaration_specifier npl_declaration

npl_declaration :       struct_definition_specifier
                        |packet_definition_specifier
                        |strength_definition_specifier
                        |bus_definition_specifier
                        |sp_definition_specifier
                        |function_definition_specifier
                        |special_func_defintion_specifier
                        |enum_defintiion_specifier
                        |program_definition_specifier
                        |table_definition_specifier
                        |register_definition_specifier
                        |parsernode_definition_specifier
                        |print_command
                        |generic_block

array_access_format : '[' postfix_expression ']'

range_access_format : '[' postfix_expression ':'postfix_expression ']'

declaration_expn :      BIT identifier ';'
                        | BIT array_access_format identifier ';'
                        | VARBIT array_access_format identifier ';'
                        | identifier identifier ';'
                        | identifier identifier array_access_format ';'
                        | BIT array_access_format identifier '==' constant ';'

field_declarator :      :FIELDS '{' field_declaration_list '}'

field_declaration_list :        declaration_expn
                                | field_declaration_list declaration_expn

key_declarator :        KEYS '{' field_declaration_list '}'
```

Наложение

```
overlay_declarator :    OVERLAYS '{' overlay_declaration_list '}'

overlay_declaration_list:       overlay_expression :
                                | overlay_declaration_list overlay_expression

overlay_expression :    identifier ':' concat_format_list ';'
```

Конкатенация сигналов

```
concat_format_list :    concat_format
                                | concat_format_list CONCAT concat_format

concat_format : identifier
                        | identifier range_access_format
```

Структура

```
struct_definition_specifier :   STRUCT identifier '{' struct_body '}'
                                | STRUCT identifier '{'field_declaration_list '}'
                                | STRUCT '{'struct_body'}'

struct_body :           field_declarator header_len_opt
                        | struct_body overlay_declarator

header_len_opt :        HEADER_LENGTH_EXP ':' expression_statement
                        | empty
```

Перечисление

```
enum_defintiion_specifier : ENUM postfix_expression args_list_format
```

Объявление шины

```
bus_definition_specifier : BUS identifier identifier ';'
```

Объявление таблицы

```
table_definition_specifier : LOGICAL_TABLE identifier '{' table_body_block '}'
table_type
table_body_block :      table_body
                        | table_body_block table_body

table_body : TABLE_TYPE ':' table_type
```

```
        | key_declarator
        | field_declarator
        | KEY_CONSTRUCT key_construct_definition_block
        | FIELDS_ASSIGN fields_assign_definition_block
        | MAXSIZE ':' constant ';'
        | MINSIZE ':' constant ';'

table_type : INDEX ';'
                | HASH ';'
                | TCAM ';'
                | ALPM ';'

table_keys_list : table_keys_expression
                | table_keys_list table_keys_expression

table_keys_expression : postfix_expression ';'

key_construct_definition_block : '{' generic_statement_list '}'

fields_assign_definition_block : '{' generic_statement_list '}'
```

Объявление регистра

```
register_definition_specifier : LOGICAL_REGISTER identifier '{' field_declarator '}'
```

Объявление пакета

```
packet_definition_specifier : packet_instance
packet_instance : PACKET identifier identifier ';'
```

Strength

```
strength_definition_specifier : strength_instance
strength_instance : STRENGTH identifier identifier ';'
```

Блок операторов

```
generic_statement_list : generic_block
                               | generic_statement_list generic_block

generic_block : statement

statement : expression_statement
                | select_statement
                | compound_statement
                | label_statement
                | header_command
                | parser_statement
                | pragma_call

compound_statement : '{' generic_statement_list '}'
```

Условные операторы

```
select_statement :      IF '(' expression ')' statement ELSE statement
                        | IF '(' expression ')' statement
                        | SWITCH '(' expression ')' statement

label_statement :       postfix_expression ':' statement
                        | DEFAULT ':' statement
                        | constant MASK constant ':' next_node
```

Выражения

```
expression_statement : expression ';'

expression :    assignment_expression
                | lookup_statement
                | parse_init
                | function_call

assignment_expression : generic_expression
                        | generic_expression assignment_operator assignment_expression

generic_expression : binary_expression

binary_expression :     unary_expression
                        | function_call
                        | binary_expression     '!=' binary_expression
                        | binary_expression     '==' binary_expression
                        | binary_expression     '&' binary_expression
                        | binary_expression     '<' binary_expression
                        | binary_expression     '<=' binary_expression
                        | binary_expression     '>=' binary_expression
                        | binary_expression     '>' binary_expression
                        | binary_expression     '|' binary_expression
                        | binary_expression     '^' binary_expression
                        | binary_expression     '&&' binary_expression
                        | binary_expression     '||' binary_expression
                        | binary_expression     '<<' binary_expression
                        | binary_expression     '>>' binary_expression
                        | binary_expression     '*' binary_expression
                        | binary_expression     '+' binary_expression
```

```
                        | binary_expression     '-' binary_expression
                        | binary_expression     '/' binary_expression
                        | binary_expression     '%' binary_expression
                        | binary_expression     '<>' binary_expression

unary_expression :      unary_operator unary_expression
                        | args_format_specifier
                        | parser_access_latest

unary_operator :        '~'
                        | '!'
                        | '|'
                        | '&'

assignment_operator : '=='

primary_expression : '(' expression')'

primary_expression :    primary_types
                                | metainfo
                                | header_position
                                | profile_type

postfix_expression :    primary_expression
                                | postfix_expression '.' identifier
                                | postfix_expression '.' metainfo
                                | postfix_expression array_access_format
                                | postfix_expression range_access_format
```

Программа

```
program_definition_specifier : PROGRAM postfix_expression '{' generic_statement_list '}'
                | PROGRAM postfix_expression '{' '}'
```

Синтаксический анализатор

```
parsernode_definition_specifier : PARSER_NODE identifier '{' generic_statement_list '}'

parser_statement :      next_node
                        | root_node
                        | parsing_done
                        | parser_field_extract

root_node : ROOT_NODE ':' constant ';'

next_node : NEXT_NODE identifier ';'

parse_init : PARSE_BEGIN '(' postfix_expression ')'

parsing_done : END_NODE':' constant';'

parser_field_extract : EXTRACT_FIELDS '(' postfix_expression ')' ';'

parser_access_latest : LATEST '.' postfix_expression
```

Обновление заголовков пакета

```
header_command :        CREATE_CHECKSUM '(' args_format_specifier ')' ';'
                | UPDATE_PACKET_LENGTH '(' args_format_specifier ')' ';'
                | ADD_HEADER '(' postfix_expression ')' ';'
                | DELETE_HEADER '(' postfix_expression ')' ';'
                | COPY_HEADER '(' postfix_expression ',' postfix_expression ')' ';'
                | REPLACE_HEADER_FIELD '(' postfix_expression ',' postfix_expression ')' ';'
```

Поиск в таблицах

```
lookup_statement : LOOKUP args_access_format
```

Функции

```
function_call :         postfix_expression '(' ')'
                        | postfix_expression args_access_format

function_definition_specifier :         FUNCTION postfix_expression '(' func_def_args ')' \
                                        '{' function_code_block '}'

func_def_args :         args_format_specifier
                        | empty

function_code_block :   statement
                                | function_code_block statement
```

Специальные функции

```
sp_definition_specifier : SFC postfix_expression postfix_expression ';'

special_func_defintion_specifier : SPECIAL_FUNCTION postfix_expression '{'
                                special_func_def_list'}'

special_func_def_list :         special_func_def
                                | special_func_def_list special_func_def

special_func_def : function_call ';'
```

Аргументы функций

```
args_access_format : '(' args_format_specifier ')'

args_type_specifier :   dir LIST postfix_expression
                        | dir STR postfix_expression

args_format_specifier :         args_type_specifier
                                | args_format_specifier ',' args_type_specifier
                                | args_list_format
                                | args_format_specifier ',' args_list_format
                                | postfix_expression
                                | args_format_specifier ',' postfix_expression
                                | args_size_dir
                                | args_format_specifier ',' args_size_dir

args_list_format :      '{' args_format_specifier '}'
                        | '{' '}'

args_def_list_format : '{' args_size_multi '}'

args_size_dir :         dir args_def_list_format
                        | dir args_size

args_size_multi :       args_size
                        | args_size_multi ',' args_size
args_size :     BIT postfix_expression
                | BIT array_access_format postfix_expression
```

Команда print

```
print_command : PRINTLN '(' STR_CONST ')'
```

Pragma

```
pragma_call : directive NPL_PRAGMA pragma_access_format

directive : PRAGMA

pragma_access_format : '(' postfix_expression ',' pragma_format_specifier ')'

pragma_format_specifier : pragma_format_specifier ',' \
                                | postfix_expression ':' postfix_expression
                                | postfix_expression
```

12. Приложение E. Директивы (@NPL_PRAGMA)

В помощь компиляторам можно задавать в прикладных программах директивы. Компиляторы FE и BE используют эти директивы для размещения и других функций. Директивы не являются частью NPL, однако для лучшего понимания здесь описан синтаксис директив и даны примеры использования.

12.1. Директивы

Директивы помогают задать желаемое поведение, которое может зависеть от оборудования. Это помогает при отображениях BE. Для директив действует ряд правил:

- директивы можно задавать в логическом файле NPL или отдельном файле;
- с директивами не связано позиционирования;
- директива должна начинаться с новой строки;
- следует использовать ключевое слово null, если объект не связан с директивой или для директивы нужно несколько объектов.

Синтаксис и ключевое слово для директив имеют вид

```
@NPL_PRAGMA(object_name, property_name:property_value);
```

Пример

Задание директивы bus_type

Если в приложении пользователь хочет определить фиксированную шину, это может иметь вид

```
bus mpls_fixed_bus_s mpls_fixed_bus;
@NPL_PRAGMA(mpls_fixed_bus, bus_type:ing_obj_fixed);
```

Задание директивы mapping

Для отображения конкретной таблицы, функции, sfc или bus_field на определённый аппаратный блок можно задать

```
function vlan_assign_functions()
@NPL_PRAGMA(vlan_assign_functions,mapping:"hw_proc_block_20")
```

13. Примеры внешних функций

Отбрасывание пакета

Для отбрасывания пакетов применяется внешняя функция packet_drop.

Таблица 27. packet_drop.

Конструкция Аргументы, опции	***Описание***
packet_drop	Отбрасывает пакет.

	bus.field_name	Задаёт имя поля, при установке которого пакет должен отбрасываться (<bus.field>). В logical_table fields_assign() это должно быть <table_field>.
	drop_code	Задаёт константу, связанную с причиной отбрасывания (constant или enum). Допустимы значения от 0 до 255, 0 означает отсутствие кода.
	strength	Задаёт приоритет или силу, связанные с отбрасыванием (константа или поле регистра).

```
packet_drop(
        <bus.field_name>,       // имя сигнала, с которым пользователь хочет связать отбрасывание
        <drop_code>,            // код причины отбрасывания
        <strength>              // приоритет причины отбрасывания
);
```

Трассировка пакетов

Для трассировки пакетов применяется внешняя функция packet_trace.

Таблица 28. packet_trace.

Конструкция Аргументы, опции		***Описание***
packet_trace		Трассировка пакета.
	bus.field_name	Задаёт имя поля, установка которого активизирует трассировку пакета (<bus.field>). В logical_table fields_assign() это должно быть <table_field>.
	trace_code	Задаёт константу, связанную с трассировкой (constant или enum). Допустимы значения от 0 до 47, 0 означает отсутствие кода.

```
packet_trace(
        <bus.field_name>,       // имя сигнала, с которым пользователь хочет связать трассировку
        <trace_code>            // код операции трассировки
);
```

Подсчёт пакетов

Для подсчёта пакетов служит внешняя функция packet_count.

Таблица 29. packet_count.

Конструкция Аргументы, опции		***Описание***
packet_count		Подсчёт пакетов.
	bus.field_name	Задаёт имя поля, установка которого активизирует подсчёт пакетов (<bus.field>). В logical_table fields_assign() это должно быть 0 или 1
	counter_id	Задаёт идентификатор счётчика (constant или enum) и может принимать значение от 1 до 63. В logical_table fields_assign() это должно быть <table_field>.

```
packet_count(
        <bus.field_name>,       // имя сигнала, с которым пользователь хочет связать подсчёт
        <counter_id>            // идентификатор счётчика
);
```

Пример

Использование packet_drop, packet_trace, packet_count

Ниже приведён пример использования count, trace и drop, выполняющий ряд задач:

- отбрасывание всех пакетов IPV4 с нулевым значением 0 с использованием drop_code 11 и priority 5;
- трассировка всех пакетов IPV4 с ttl = 1;
- подсчёт всех пакетов IPV4 с ttl = 2

```
struct cond_bus_s {
        bit     ttl_0;
        bit     ttl_1;
        bit     ttl_2;
}

bus cond_bus_s cond_bus;
#define TTL0 11
function func_ttl_proc () {
        if (header_ipv4.ttl == 0) {
                cond_bus.ttl_0 = 1;
        }
        if (header_ipv4.ttl == 1) {
                cond_bus.ttl_1 = 1;
        }
        if (header_ipv4.ttl == 2) {
                cond_bus.ttl_2 = 1;
        }
        packet_drop(cond_bus.ttl_0, TTL0, 5);
        packet_trace(cond_bus.ttl_1, 3);
        packet_count(cond_bus.ttl_2, 1);
}

program ipv4() {
        ...
        func_ttl_proc();
        ...
}
```

Перевод на русский язык
Николай Малых
nmalykh@protokols.ru