Overview

This lab starts by describing how to define custom headers in a P4 program. It then explains how to implement a simple parser that parses the defined headers. The lab further shows how to track the parsing states of a packet inside the software switch.

Objectives

By the end of this lab, students should be able to:

1. Define custom headers in a P4 program.
2. Understand how the parser transitions between states and how it extracts the headers from the packets.
3. Implement a simple parser in P4.
4. Trace the parsed states when a packet enters to the switch.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Defining the headers.
4. Section 4: Parser implementation.
5. Section 5: Loading the P4 program.
6. Section 6: Configuring switch s1.
7. Section 7: Testing and verifying the P4 program.
8. Section 8: Augmenting the P4 program to parse IPv6.
9. Section 9: Testing and verifying the augmented P4 program.

1 Introduction

1.1 Program headers and definitions

For several decades, the networking industry operated in a bottom-up approach. At the bottom of the system are the fixed-function Application Specific Integrated Circuits (ASICs), which enforce protocols, features, and processes available in the switch. Programmers and operators are limited to these capabilities when building their systems. Consequently, systems have features defined by ASIC vendors that are rigid and may not fit the network operators' needs. Programmable switches and P4 represent a disruption of the networking industry by enabling a top-down approach for the design of network applications. With this approach, the programmer or network operator can precisely describe features and how packets are processed in the ASIC, using a high-level language, P4.

With the Protocol Independent Switch Architecture (PISA)[1], the programmer defines the headers and corresponding parser as well as actions executed in the match-action pipeline and the deparser. The programmer has the flexibility of defining custom headers (i.e., a header not standardized). Such capability is not available in non-programmable devices.

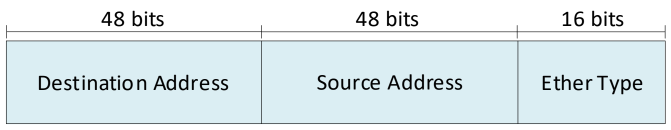


Figure 1. Ethernet header.

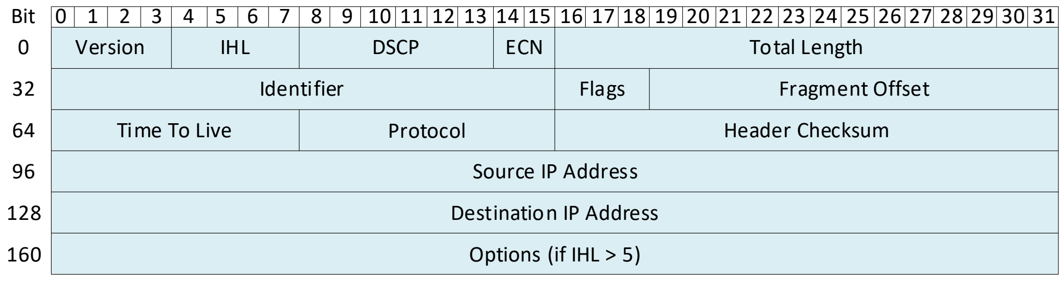


Figure 2. IPv4 header.

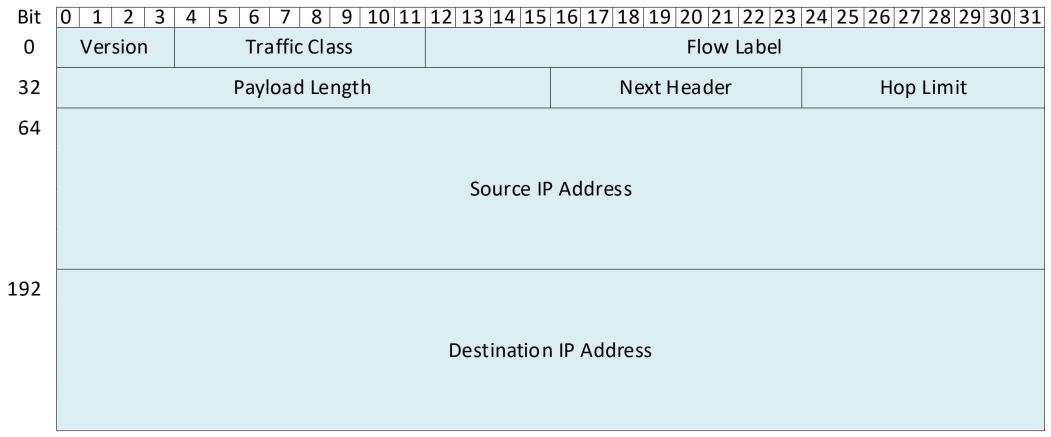
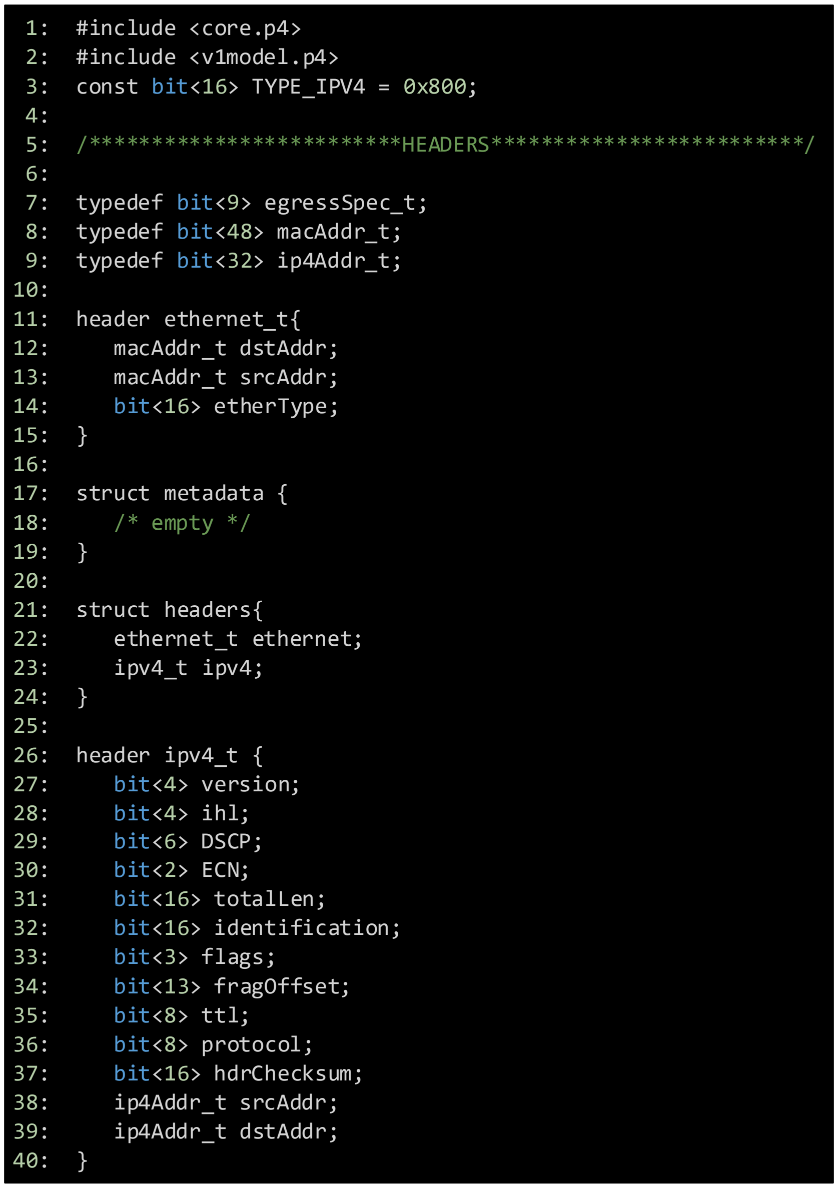


Figure 3. IPv6 header.

Figure 4 shows an excerpt of a P4 program where the headers are defined. This is typically written at the top of the program before the parsing starts. We can see that the programmer defined a header corresponding to Ethernet (lines 11-15). The Ethernet header fields are shown in Figure 1.

The programmer also defined an IPv4 header (lines 26-40). The IPv4 header format is shown in Figure 2 and the IPv6 header is shown in Figure 3.



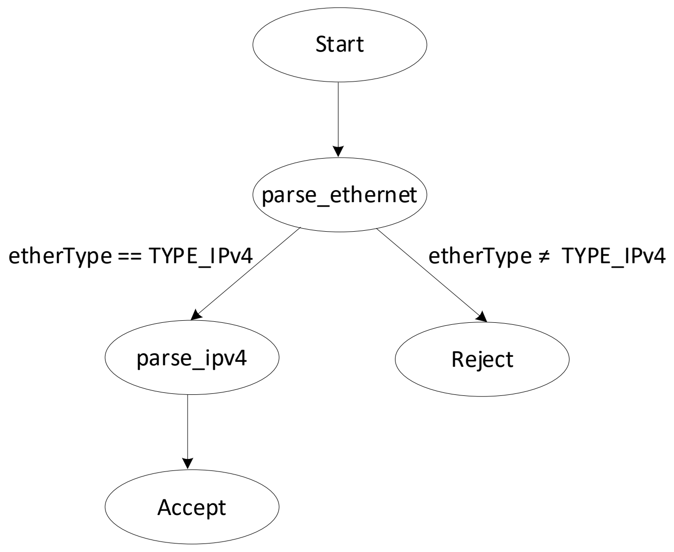
```
 1:  #include <core.p4>
 2:  #include <v1model.p4>
 3:  const bit<16> TYPE_IPV4 = 0x800;
 4:
 5:  /************************HEADERS************************/
 6:
 7:  typedef bit<9> egressSpec_t;
 8:  typedef bit<48> macAddr_t;
 9:  typedef bit<32> ip4Addr_t;
10:
11:  header ethernet_t{
12:     macAddr_t dstAddr;
13:     macAddr_t srcAddr;
14:     bit<16> etherType;
15:  }
16:
17:  struct metadata {
18:     /* empty */
19:  }
20:
21:  struct headers{
22:     ethernet_t ethernet;
23:     ipv4_t ipv4;
24:  }
25:
26:  header ipv4_t {
27:     bit<4> version;
28:     bit<4> ihl;
29:     bit<6> DSCP;
30:     bit<2> ECN;
31:     bit<16> totalLen;
32:     bit<16> identification;
33:     bit<3> flags;
34:     bit<13> fragOffset;
35:     bit<8> ttl;
36:     bit<8> protocol;
37:     bit<16> hdrChecksum;
38:     ip4Addr_t srcAddr;
39:     ip4Addr_t dstAddr;
40:  }
```

Figure 4. Program headers and definitions.

The code starts by including the *core.p4* file (line 1) which defines some common types and variables used in all P4 programs. For instance, the `packet_in` and `packet_out` extern types which represent incoming and outgoing packets, respectively, are declared in *core.p4*[2]. Next, the *v1model.p4*[3] file is included (line 2) to define the V1Model architecture[4] and all its externs used when writing P4 programs. Line 3 creates a 16-bit constant `TYPE_IPV4` with the value 0x800. This means that `TYPE_IPV4` can be used later in the P4 program to reference the value 0x800. The typedef declarations (lines 7 - 9) are used to assign alternative names to types. Subsequently, the headers and the metadata structs that will be used in the program are defined. These headers are customized depending on how the programmer wants the packets to be parsed. The program in Figure 4 defines the Ethernet header (lines 11-15) and the IPv4 header (lines 26-40). The declarations inside each header are usually written after referring to the standard specifications of the protocol. Note in the `ethernet_t` header the `macAddr_t` is used rather than using a 48-bit field. Lines 17 - 19 show how to declare user-defined metadata, which are passed from one block to another as the packet propagates through the architecture. For simplicity, this program does not require any user metadata.

1.2 Programmable parser

The programmable parser permits the programmer to describe how the switch will process the packet. The parser de-encapsulates the headers, converting the original packet into a parsed representation of the packet. The parser can be represented as a state machine without cycles (direct acyclic graph), with one initial state (start) and two final states (accept or reject).



(a)

```
 1:  /*************************HEADERS*************************/
 2:  parser MyParser( packet_in packet, out headers hdr,
 3:                  inout metadata meta,
 4:                  inout standard_metadata_t standard_metadata ){
 5:     state start {
 6:         transition parse_ethernet;
 7:     }
 8:     state parse_ethernet {
 9:         packet.extract(hdr.ethernet);
10:         transition select(hdr.ethernet.etherType) {
11:             TYPE_IPV4: parse_ipv4;
12:             default: reject;
13:         }
14:     }
15:     state parse_ipv4 {
16:         packet.extract(hdr.ipv4);
17:         transition accept;
18:     }
19:  }
```

(b)

Figure 5. Example of a parser. (a) Graphical representation of the parser. (b) In P4, the parser always starts with the initial state called `start`. First, we transition unconditionally to `parse_ethernet`. Then, we can create some conditions to direct the parser. Finally, when we transition to the `accept` state, the packet is moved to the ingress block of the pipeline. A packet that reaches the `reject` state will be dropped.

Figure 5a shows the graphical representation of the parser and Figure 5b its corresponding P4 code. Note that packet is an instance of the `packet_in` extern (specific to V1Model) and is passed as a parameter to the parser. The `extract` method associated with the packet extracts N bits, where N is the total number of bits defined in the corresponding header (for example, 112 bits for Ethernet). Afterwards, the `etherType` field of the Ethernet header is examined using the select statement, and the program branches to the `parse_ipv4` state if the `etherType` field corresponds to IPv4. The state transitions to the `reject` if it is not an IPv4 header, as shown in the figure above (Line 12). In the `parse_ipv4` state, the IPv4 header is extracted, and the program unconditionally transitions to the `accept` state.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit.

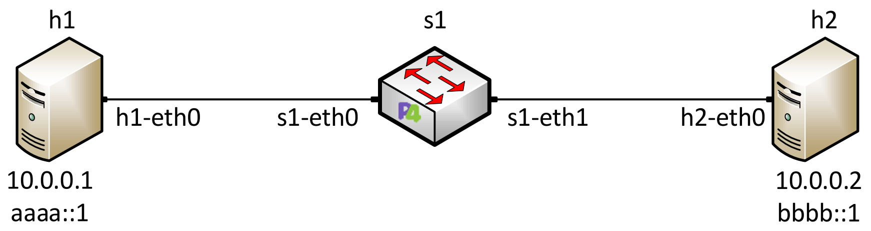


Figure 6. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

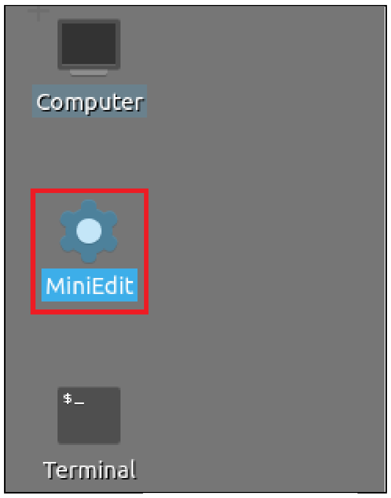
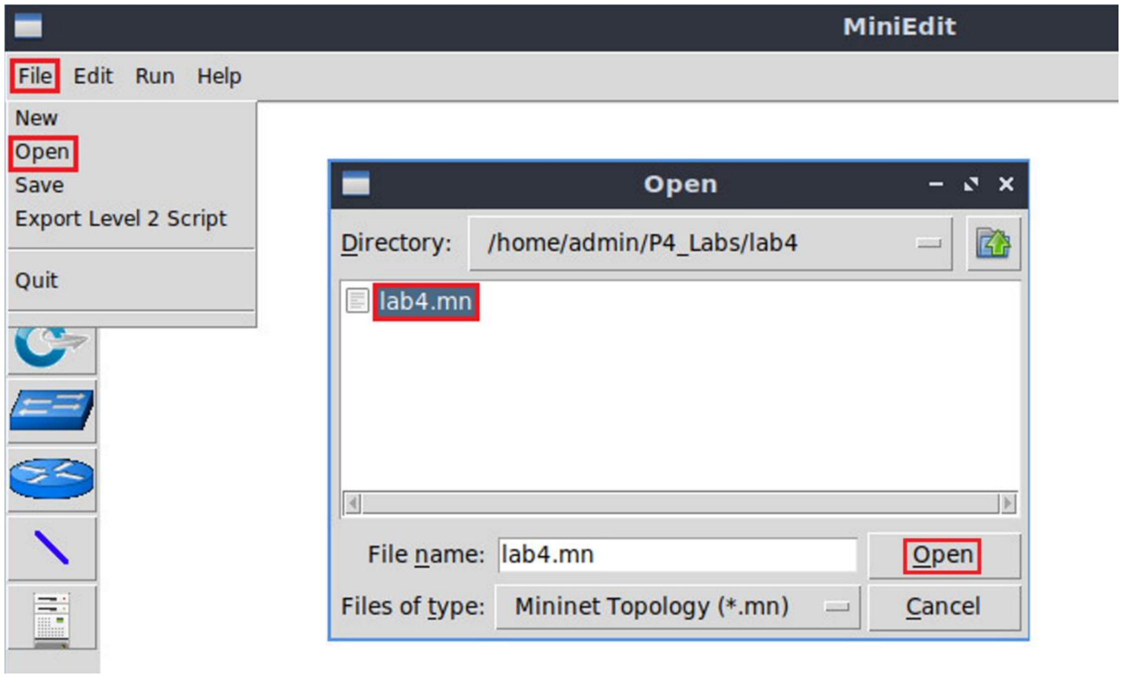


Figure 7. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab4* folder and search for the topology file called *lab4.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4_Labs/lab4

lab4.mn

File name: lab4.mn

Files of type: Mininet Topology (*.mn)

Figure 8. MiniEdit's *Open* dialog.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

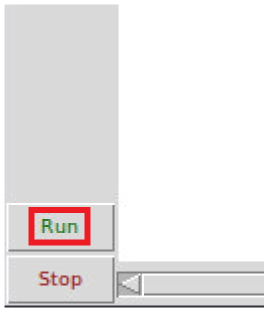


Figure 9. Running the emulation.

2.1 Starting host h1 and host h2

Step 1. Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

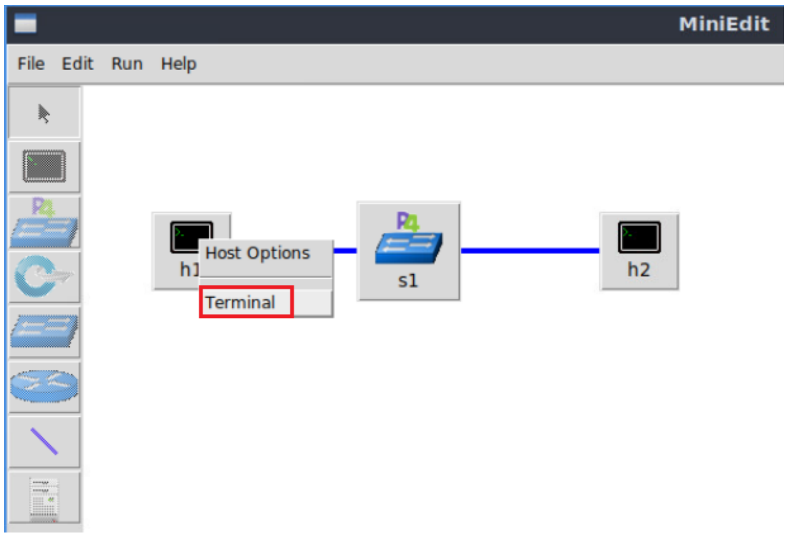


Figure 10. Opening a terminal on host h1.

3 Defining the program’s headers

This section demonstrates how to define custom headers in a P4 program. It also shows how to use constants and typedefs to make the program more readable.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the icon located on the desktop.

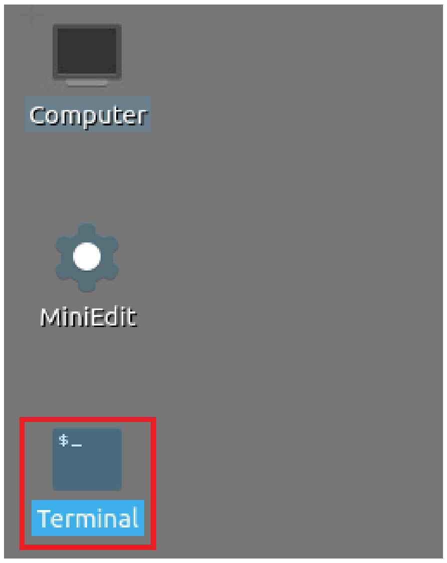
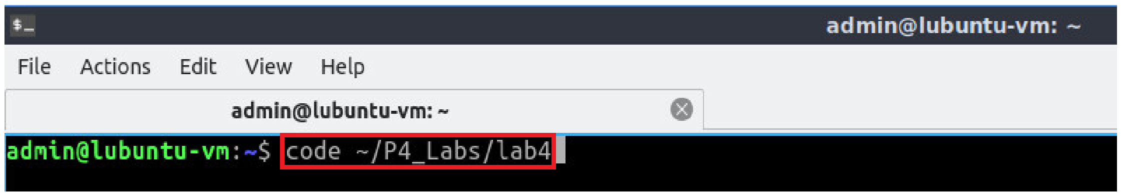


Figure 11. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). A CLI is a program that takes commands from the keyboard and sends them to the operating system to perform.

Step 2. In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab4
```

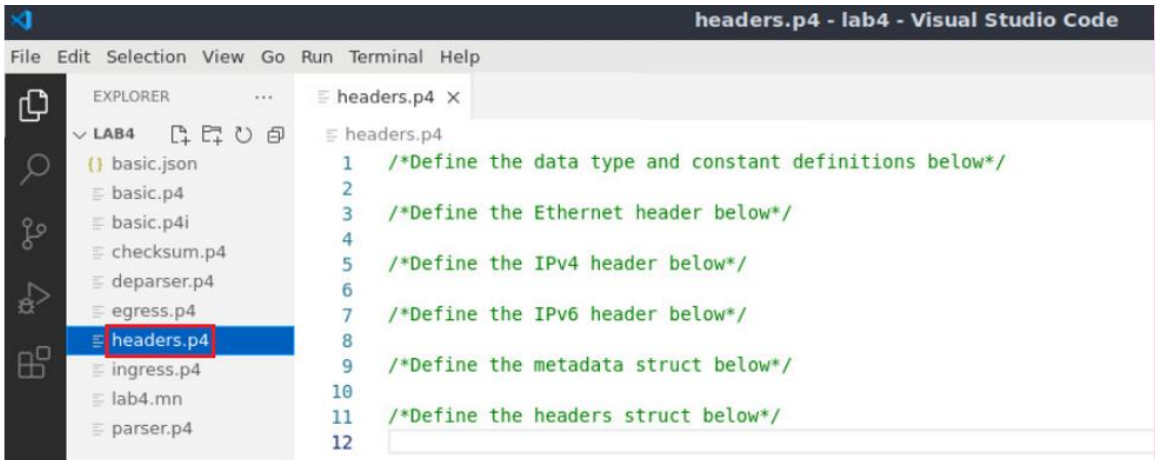


```
admin@lubuntu-vm:~$ code ~/P4_Labs/lab4
```

Figure 12. Launching the editor and opening the lab4 directory.

3.2 Coding header's definitions into the *headers.p4* file

Step 1. Click on the *headers.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.



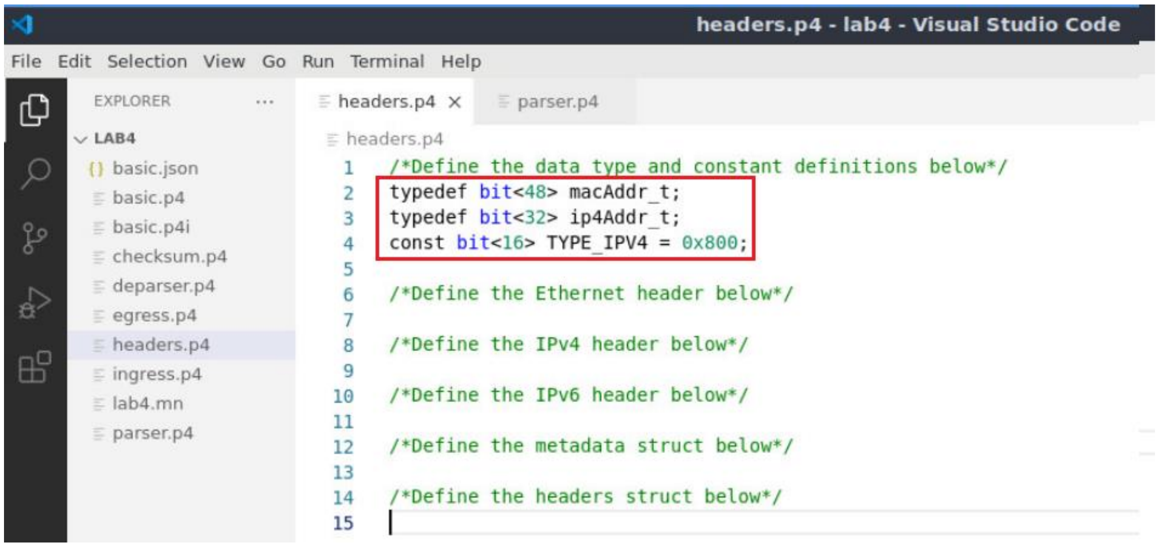
```
/*Define the data type and constant definitions below*/

/*Define the Ethernet header below*/

/*Define the IPv4 header below*/

/*Define the IPv6 header below*/

/*Define the metadata struct below*/

/*Define the headers struct below*/

```

Figure 13. Inspecting the *headers.p4* file.

We can see that the *headers.p4* is empty and we have to fill it.

Step 2. We will start by defining some typedefs and constants. Write the following in the *headers.p4* file.

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;
```



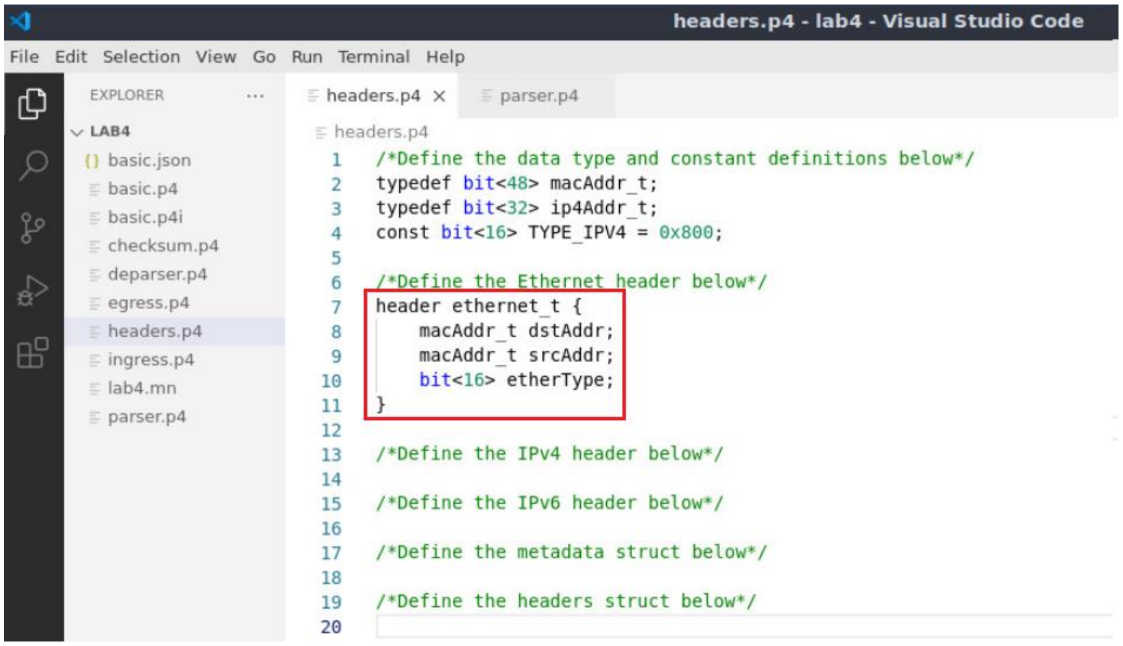
```
1  /*Define the data type and constant definitions below*/
2  typedef bit<48> macAddr_t;
3  typedef bit<32> ip4Addr_t;
4  const bit<16> TYPE_IPV4 = 0x800;
5
6  /*Define the Ethernet header below*/
7
8  /*Define the IPv4 header below*/
9
10 /*Define the IPv6 header below*/
11
12 /*Define the metadata struct below*/
13
14 /*Define the headers struct below*/
15
```

Figure 14. Data types and constant definitions.

In the figure above the typedef declarations used (lines 2 - 3) are used to assign alternative names to types. Here we are saying that `macAddr_t` can be used instead of `bit<48>`, and `ip4Addr_t` instead of `bit<32>`. We will use those typedefs when defining the headers. Line 4 shows how to define a constant with the name `TYPE_IPV4` and a value of `0x800`. We will use this value in the parser implementation.

Step 3. Now we will define the Ethernet header. Add the following code to *the headers.p4* file.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}
```



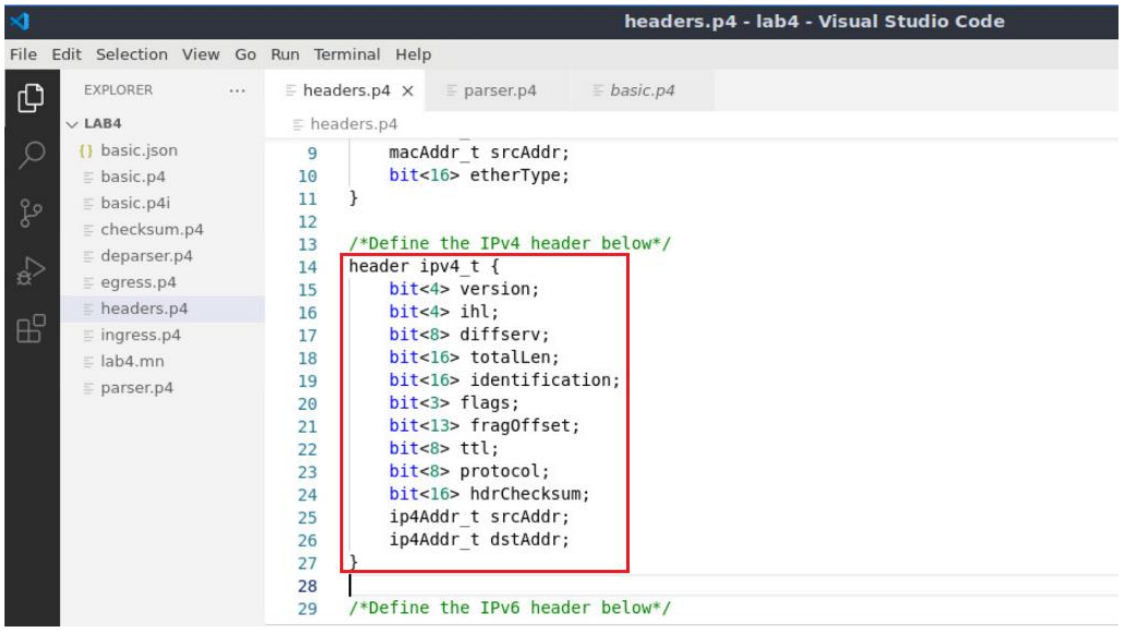
```
/*Define the data type and constant definitions below*/
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;

/*Define the Ethernet header below*/
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}

/*Define the IPv4 header below*/

/*Define the IPv6 header below*/

/*Define the metadata struct below*/

/*Define the headers struct below*/
```

Figure 15. Adding the Ethernet header definition.

Note how we used the typedef `macAddr_t` which corresponds to `bit<48>` when defining the destination MAC address field (`dstAddr`) and the source MAC address field (`srcAddr`).

Step 4. Now we will define the IPv4 header. Add the following to the *headers.p4* file.

```
header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```



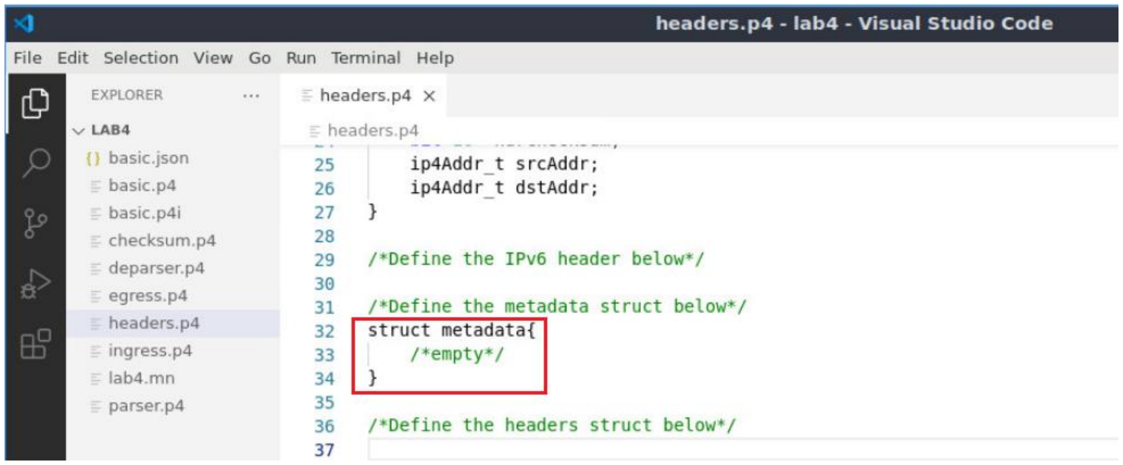
```
    macAddr_t srcAddr;
    bit<16> etherType;
}

/*Define the IPv4 header below*/
header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffserv;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/
```

Figure 16. Adding the IPv4 header definition.

Consider the figure above. Note how we used the typedef `ip4Addr_t` which corresponds to `bit<32>` when defining the source IP address field (`srcAddr`) and the destination IP address field (`dstAddr`). Also, note how we are mapping the fields to those defined in the standard IPv4 header (see Figure 3).

Step 5. Now we will create a struct to represent our metadata. Metadata are passed from one block to another as the packet propagates through the architecture. For simplicity, this program does not require any user metadata, and hence we will define it as empty with no fields. Add the following to the *headers.p4* file.

```
struct metadata {
    /* empty */
}
```

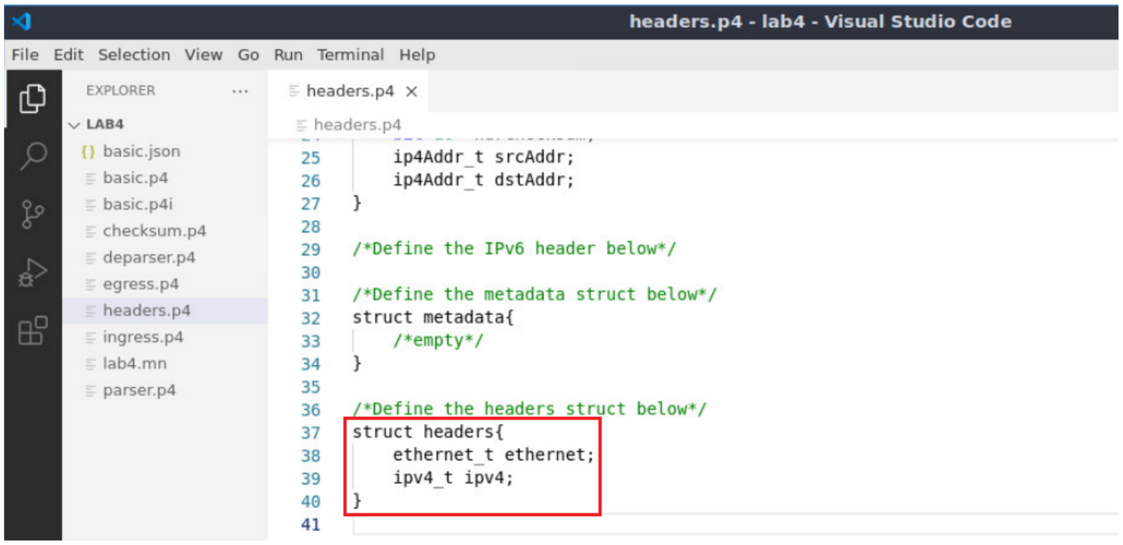


```
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/

/*Define the metadata struct below*/
struct metadata{
    /*empty*/
}

/*Define the headers struct below*/
```

Figure 17. Adding the metadata structures.

Step 6. Now we will create a struct to contain our headers (Ethernet and IPv4). Append the following code to the *headers.p4* file.

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```



```
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/

/*Define the metadata struct below*/
struct metadata{
    /*empty*/
}

/*Define the headers struct below*/
struct headers{
    ethernet_t ethernet;
    ipv4_t ipv4;
}
```

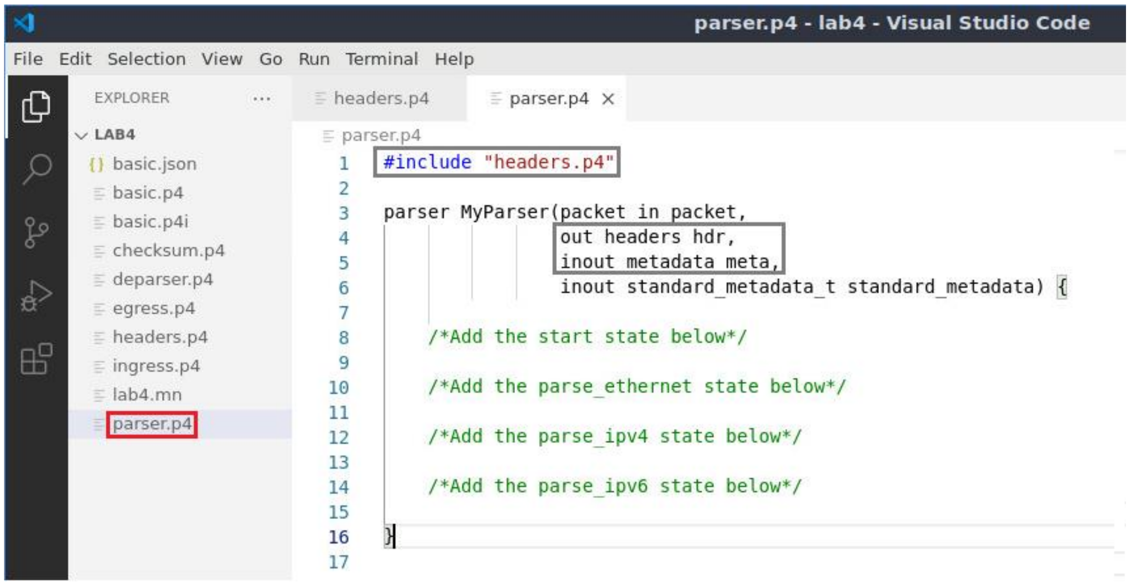
Figure 18. Appending the headers' data structure to the *headers.p4* file.

Step 7. Save the changes by pressing `Ctrl+s`.

4 Parser Implementation

Now it is time to define how the parser works.

Step 1. Click on the *parser.p4* file to display the content of the file.



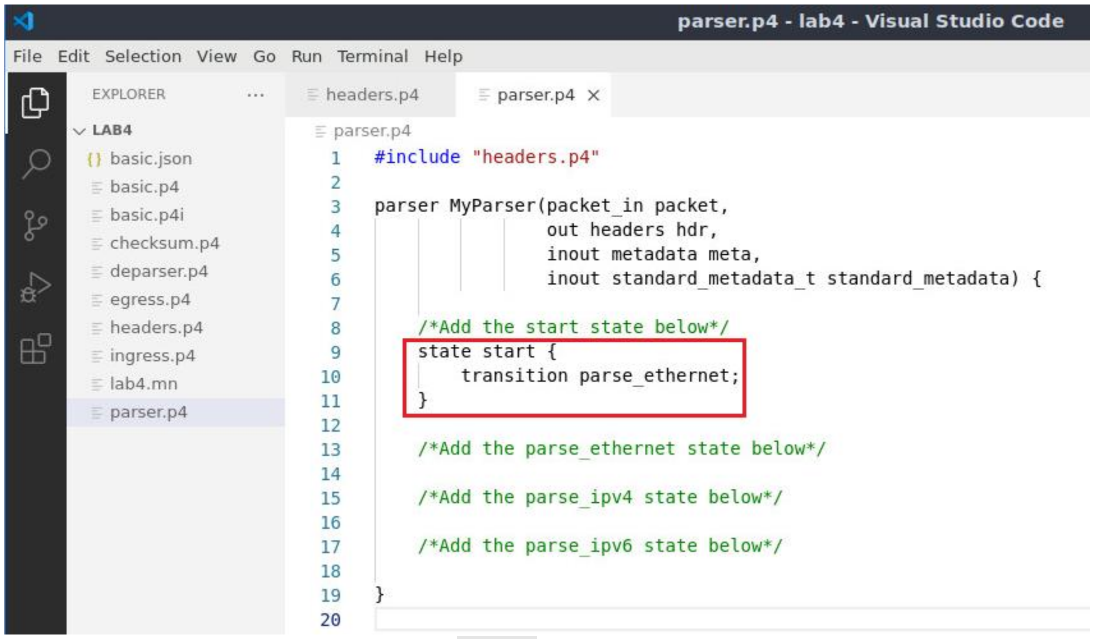
```
#include "headers.p4"

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    /*Add the start state below*/

    /*Add the parse_ethernet state below*/

    /*Add the parse_ipv4 state below*/

    /*Add the parse_ipv6 state below*/

}
```

Figure 19. Inspecting the *parse.p4* file.

We can see that the *headers.p4* file that we just filled is included here in the parser. The file also includes a starter code which declares a parser named *MyParser*. Note how the headers and the metadata structs that we defined previously are passed as parameters to the parser.

Step 2. Add the `start` state inside the parser by inserting the following code.

```
state start {
     transition parse_ethernet;
}
```



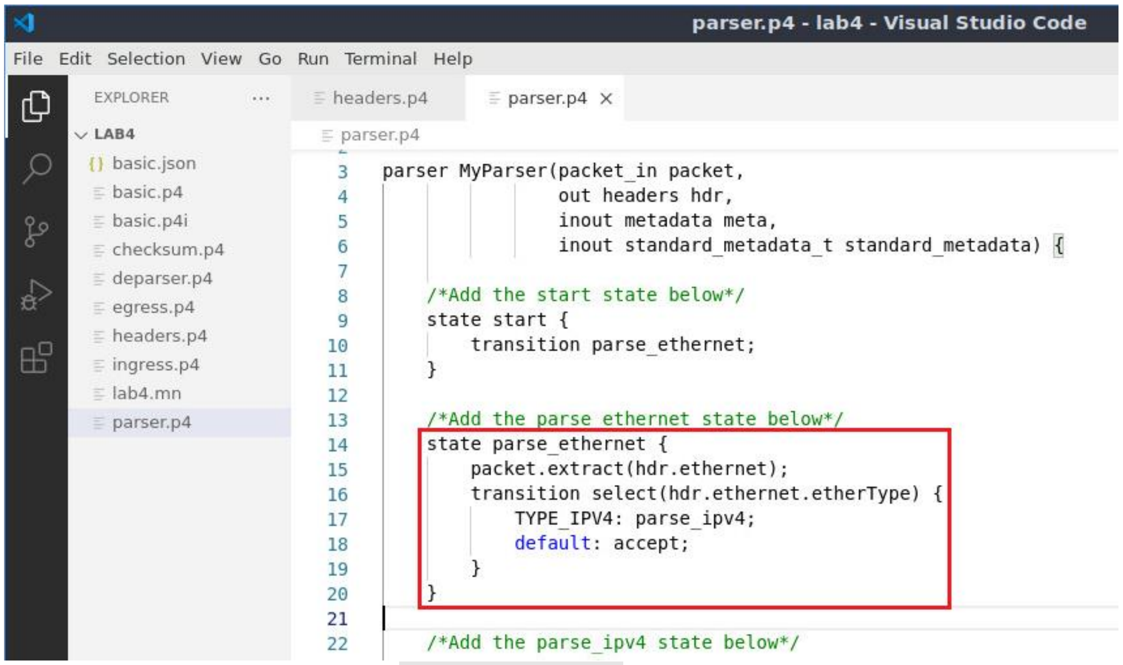
```
#include "headers.p4"

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    /*Add the start state below*/
    state start {
        transition parse_ethernet;
    }

    /*Add the parse_ethernet state below*/

    /*Add the parse_ipv4 state below*/

    /*Add the parse_ipv6 state below*/

}
```

Figure 20. Adding `start` state to the *parser.p4* file.

The `start` state is the state where the parser begins parsing the packet. Here we are transitioning unconditionally to the `parse_ethernet` state.

Step 3. Add the `parse_ethernet` state inside the parser by inserting the following code.

```
state parse_ethernet {
   packet.extract(hdr.ethernet);
   transition select(hdr.ethernet.etherType) {
      TYPE_IPV4: parse_ipv4;
      default: accept;
   }
}
```



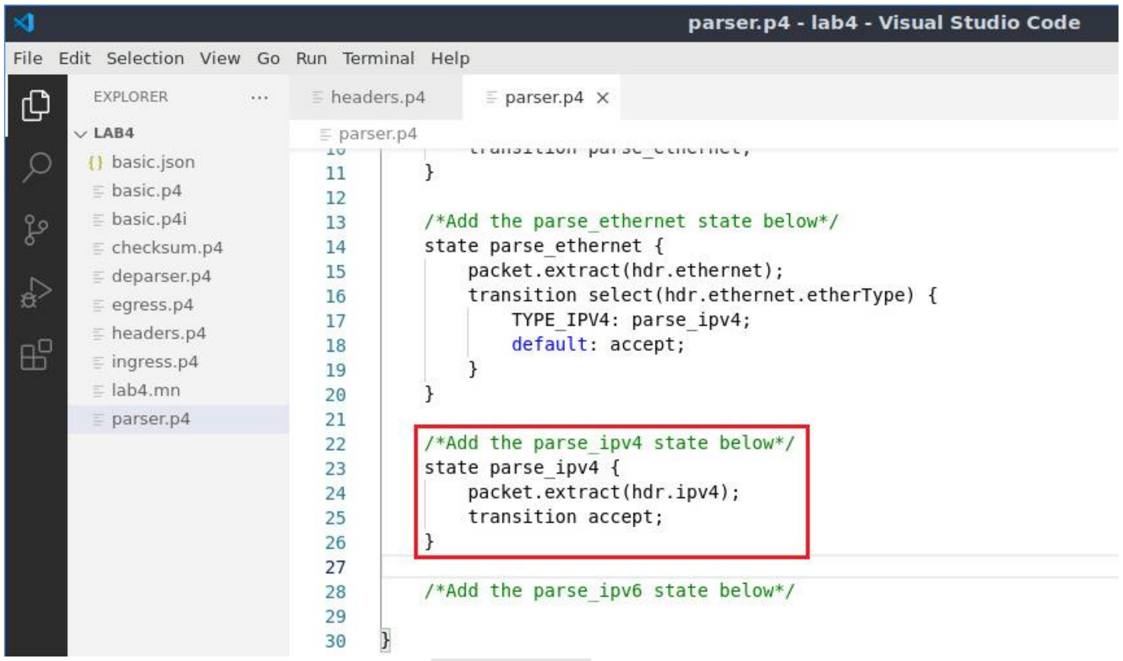
```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    /*Add the start state below*/
    state start {
        transition parse_ethernet;
    }

    /*Add the parse ethernet state below*/
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    /*Add the parse_ipv4 state below*/
```

Figure 21. Adding `parse_ethernet` state to the *parser.p4* file.

The `parse_ethernet` state extracts the Ethernet header and checks for the value of the header field `etherType`. Note how we reference a header field by specifying the header to which that field belongs (i.e., `hdr.ethernet.etherType`). If the value of `etherType` is `TYPE_IPV4` (which corresponds to 0x800 as defined previously), the parser transitions to the `parse_ipv4` state. Otherwise, the execution of the parser terminates.

Step 4. Add the `parse_ipv4` state inside the parser by inserting the following code.

```
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```



```
    transition parse_ethernet;
    }

    /*Add the parse_ethernet state below*/
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    /*Add the parse_ipv4 state below*/
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

    /*Add the parse_ipv6 state below*/

}
```

Figure 22. Adding `parse_ipv4` state to the *parser.p4* file.

The `parse_ipv4` state extracts the IPv4 header and terminates the execution of the parser.

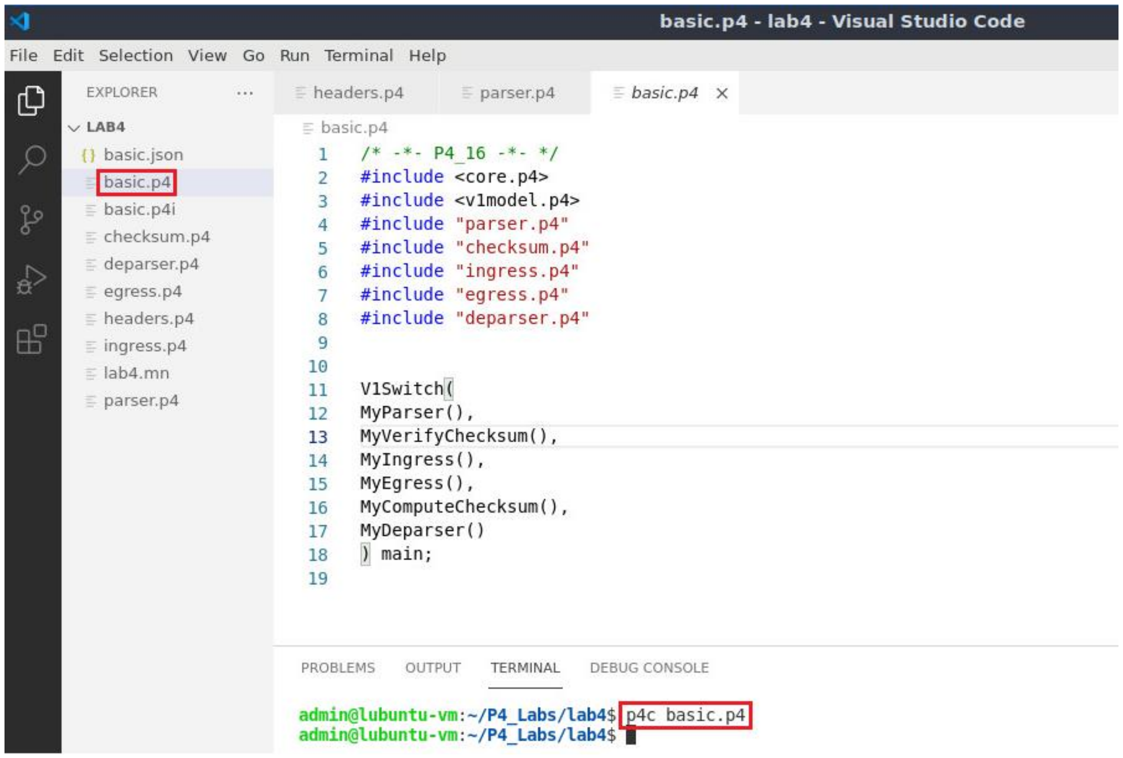
Step 5. Save the changes to the file by pressing `Ctrl + s`.

5 Loading the P4 program

5.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```



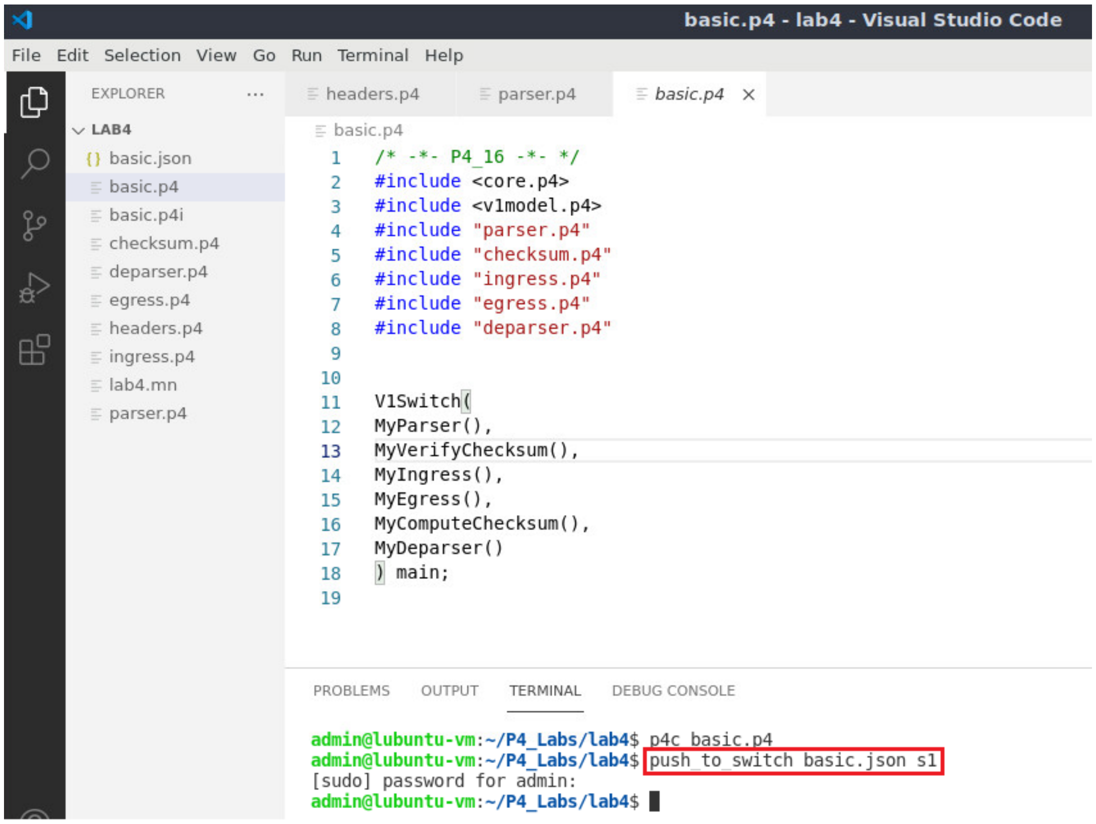
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 23. Compiling the code.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the *p4c* compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 24. Pushing the P4 program to switch s1.

5.2 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.

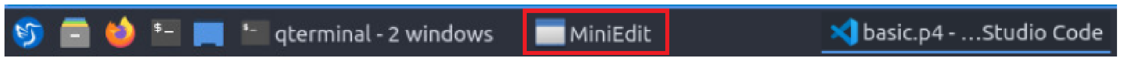


Figure 25. Maximizing the MiniEdit window.

Step 2. Right-click on the P4 switch icon in MiniEdit and start the *Terminal*.

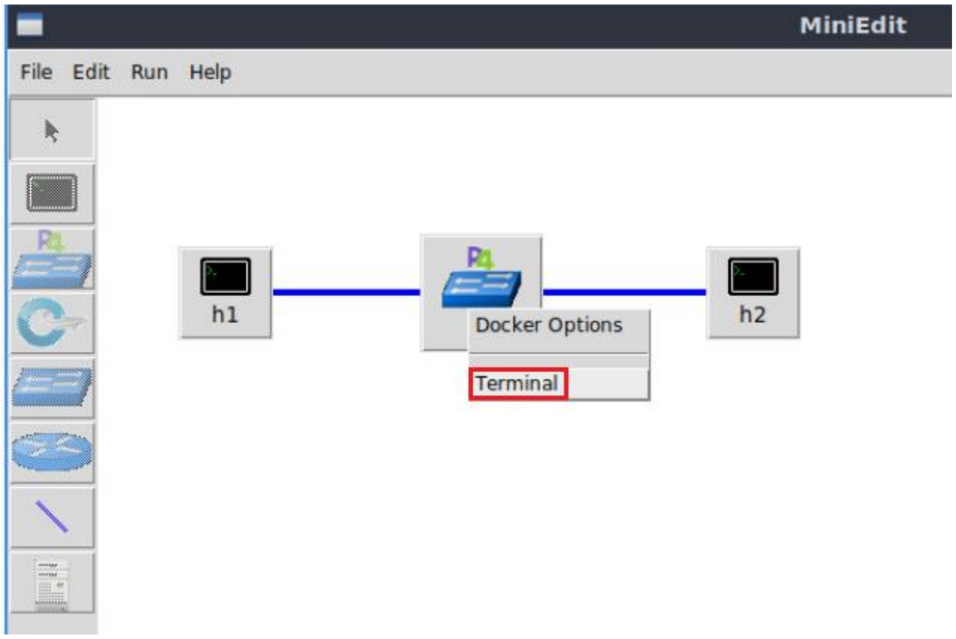
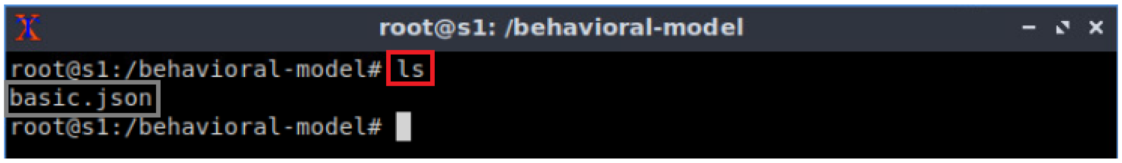


Figure 26. Starting the terminal on the switch.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

Step 3. Issue the command `ls` on the terminal of the switch s1 that was opened in the previous step.

```
ls
```



```
root@s1: /behavioral-model
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

Figure 27. Displaying the contents of the current directory in the switch s1.

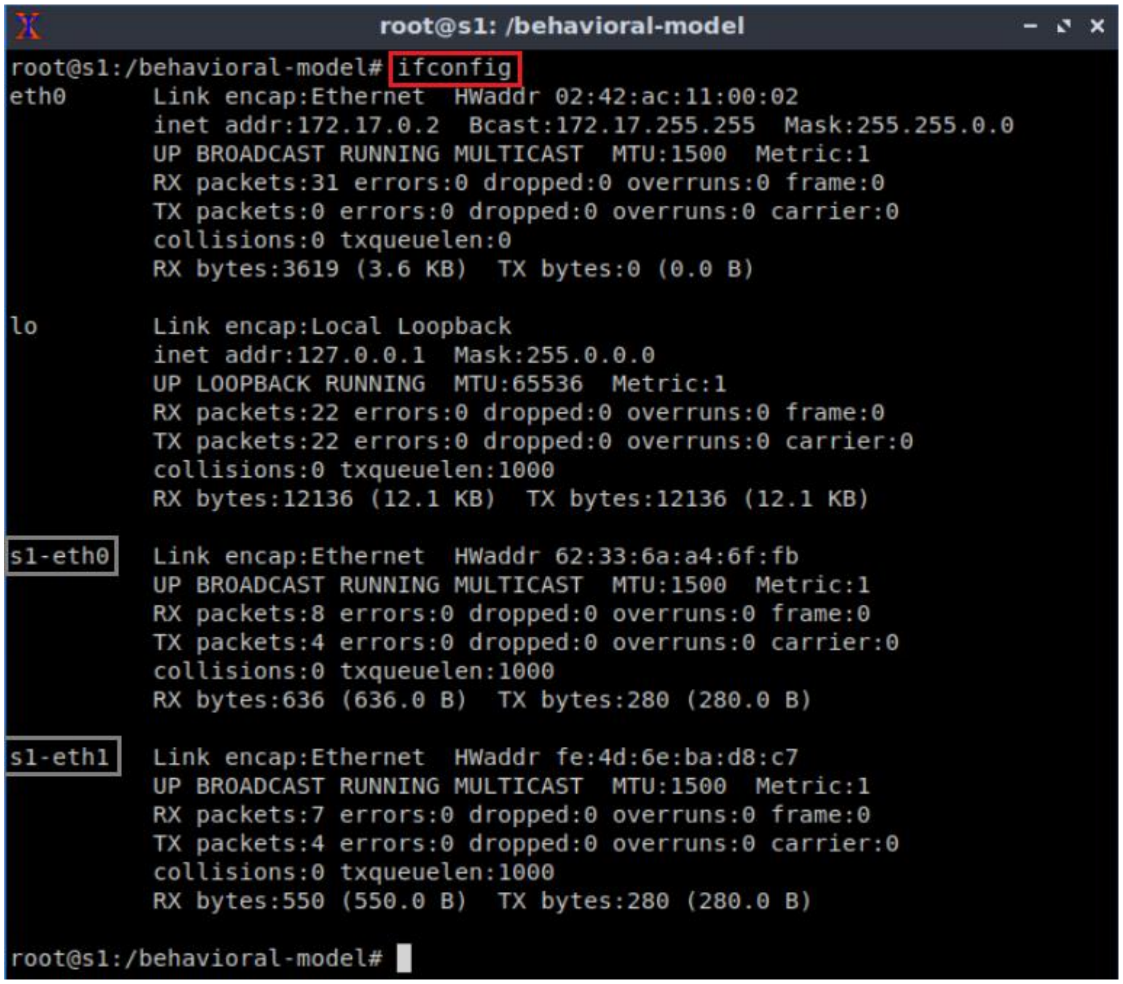
We can see that the switch contains the *basic.json* file that was pushed previously after compiling the P4 program.

6 Configuring switch s1

6.1 Mapping P4 program's ports

Step 1. Issue the following command on switch s1 terminal to display the interfaces.

```
ifconfig
```



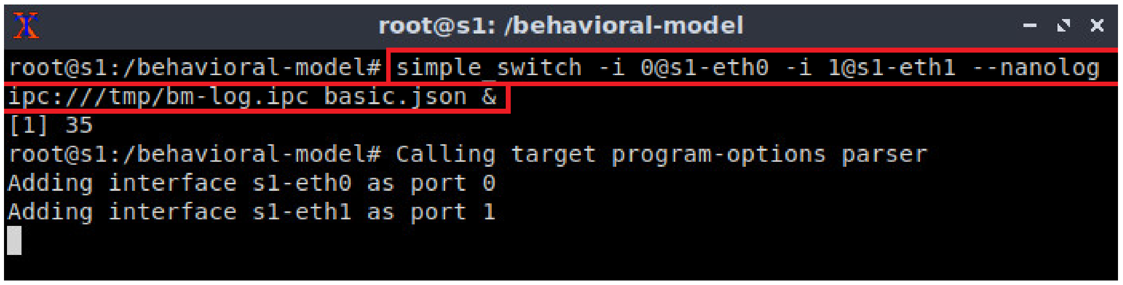
```
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3619 (3.6 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12136 (12.1 KB)  TX bytes:12136 (12.1 KB)

s1-eth0   Link encap:Ethernet  HWaddr 62:33:6a:a4:6f:fb
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:636 (636.0 B)  TX bytes:280 (280.0 B)

s1-eth1   Link encap:Ethernet  HWaddr fe:4d:6e:ba:d8:c7
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:550 (550.0 B)  TX bytes:280 (280.0 B)

root@s1:/behavioral-model#
```

Figure 28. Displaying switch s1 interfaces.

We can see that the switch has the interfaces *s1-eth0* and *s1-eth1*. The interface *s1-eth0* on the switch s1 connects host h1. The interface *s1-eth1* on the switch s1 connects host h2.

Step 2. Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog ipc:///tmp/bm-log.ipc
basic.json &
```



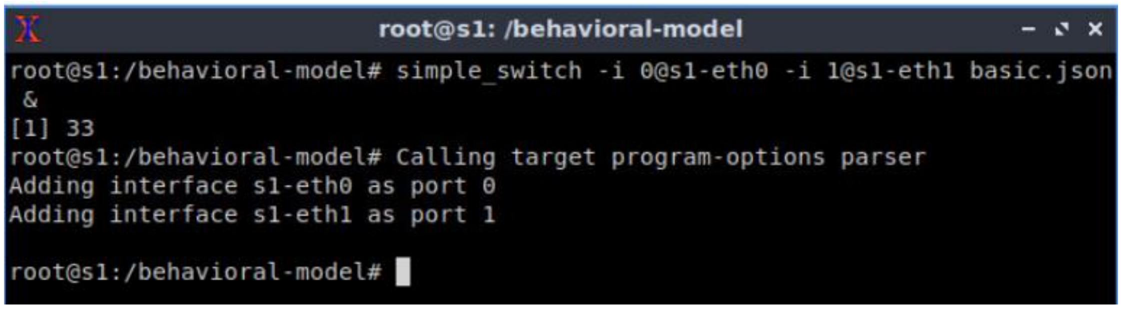
```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 35
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 29. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

The `--nanolog` parameter is used to instruct the switch daemon that we want to see the logs of the switch.

6.2 Loading the rules to the switch

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

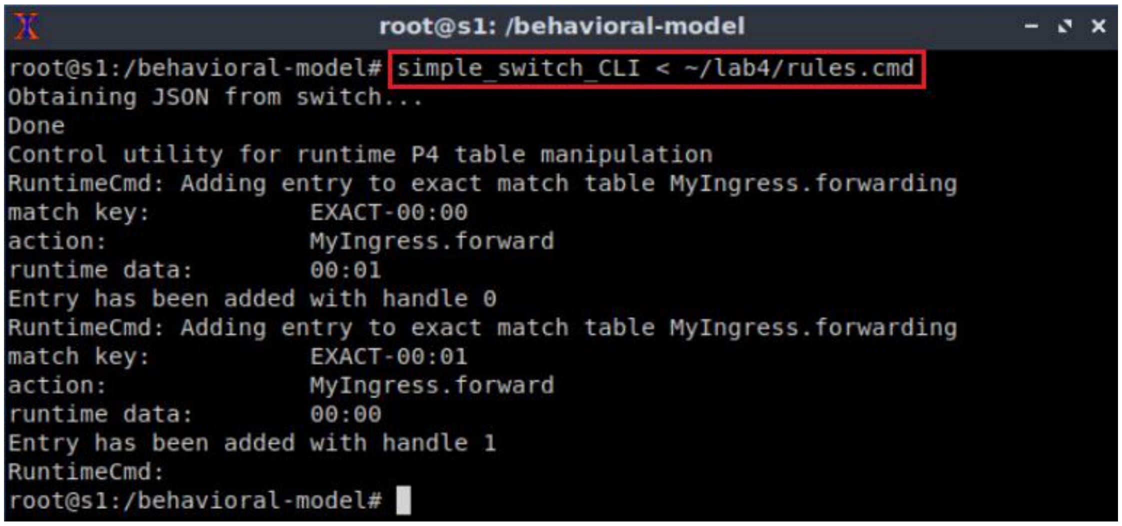


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model#
```

Figure 30. Returning to switch s1 CLI.

Step 2. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab4/rules.cmd
```



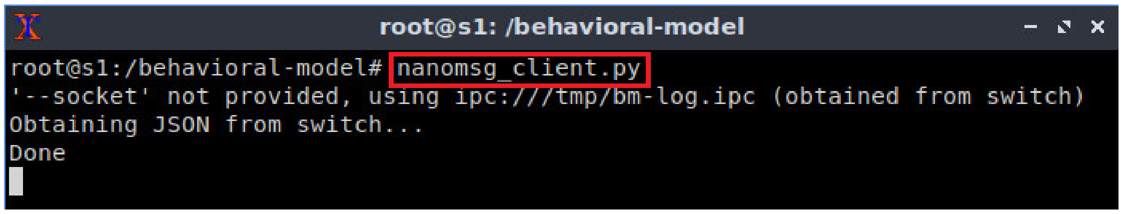
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab4/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 31. Populating the forwarding table into switch s1.

7 Testing and verifying the P4 program

Step 1. Type the following command to initiate the `nanolog` client that will display the switch logs.

```
nanomsg_client.py
```

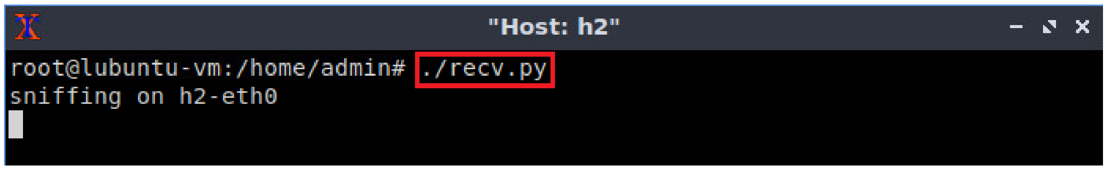


```
root@s1: /behavioral-model
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 32. Displaying switch s1 logs.

Step 2. On host h2's terminal, type the command below so that the host starts listening for packets.

```
./recv.py
```

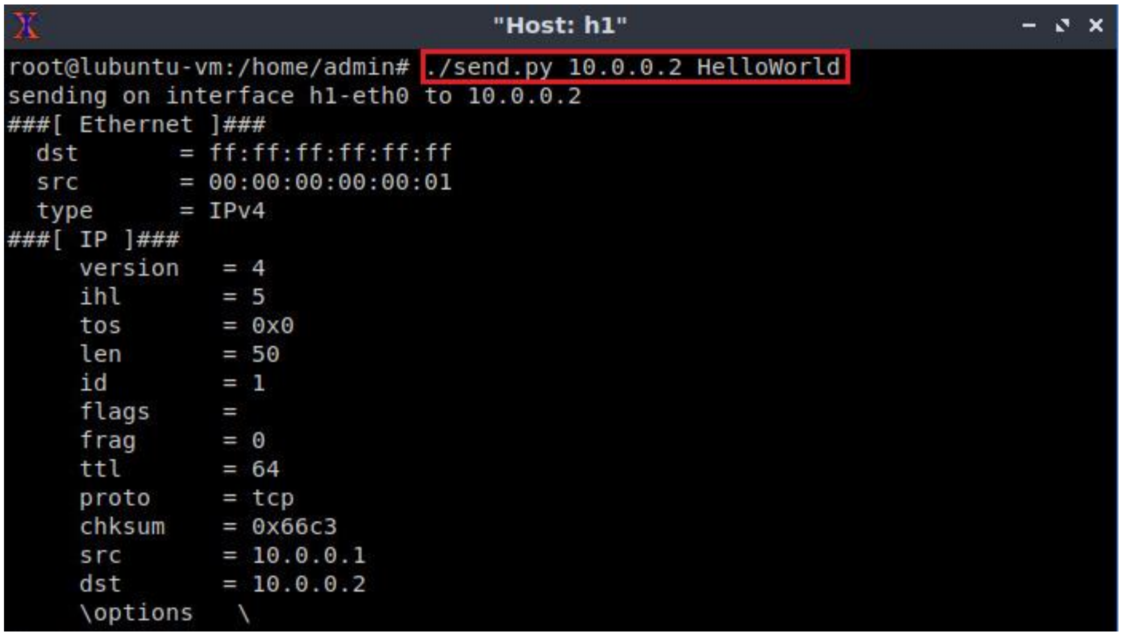


```
"Host: h2"
root@lubuntu-vm:/home/admin# ./recv.py
sniffing on h2-eth0
```

Figure 33. Listening for incoming packets in host h2.

Step 3. On host h1's terminal, type the following command to send a packet to host h2.

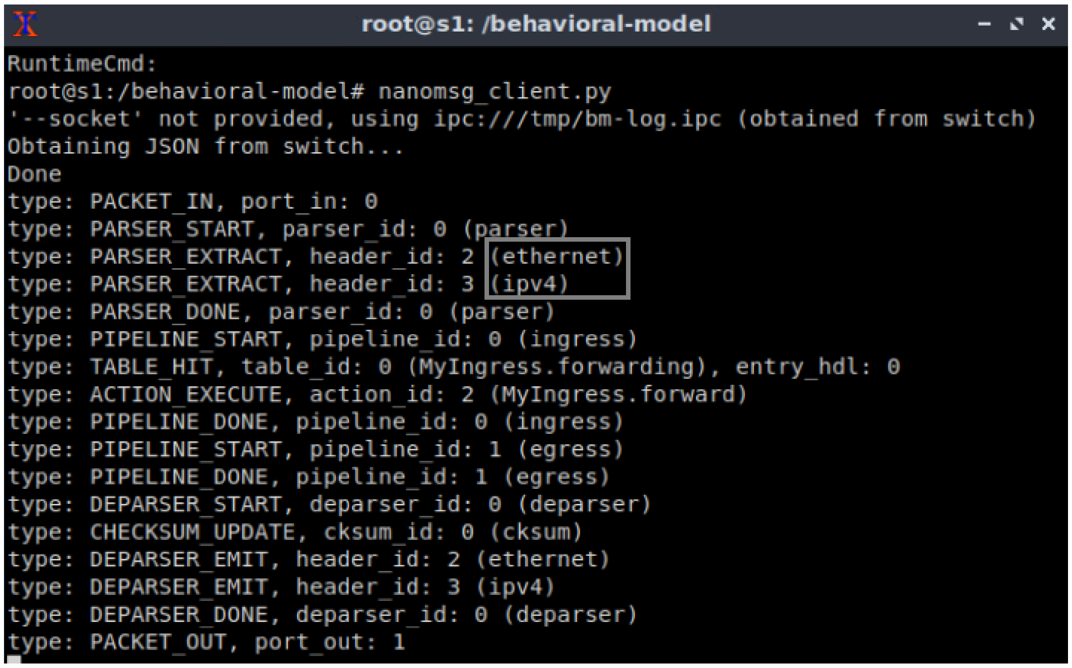
```
./send.py 10.0.0.2 HelloWorld
```



```
"Host: h1"
root@lubuntu-vm:/home/admin# ./send.py 10.0.0.2 HelloWorld
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 50
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x66c3
     src       = 10.0.0.1
     dst       = 10.0.0.2
     \options   \
```

Figure 34. Sending a test packet from host h1 to host h2.

Step 4. Inspect the logs on switch s1 terminal.



```
root@s1: /behavioral-model
RuntimeCmd:
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 35. Inspecting the logs in switch s1.

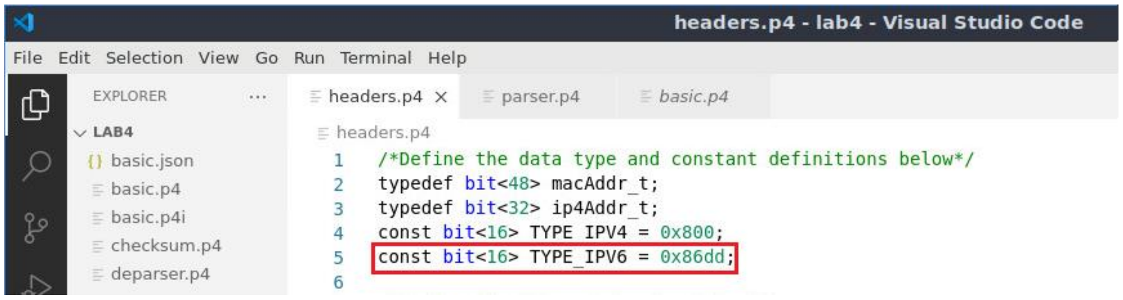
The figure above shows that the Ethernet and IPv4 header are extracted.

8 Augmenting the P4 program to parse IPv6

Now we will augment the program to parse IPv6 packets. Figure 4 shows the IPv6 header fields.

Step 1. Go back to the *headers.p4* file and add the following constant definition.

```
const bit<16> TYPE_IPV6 = 0x86dd;
```



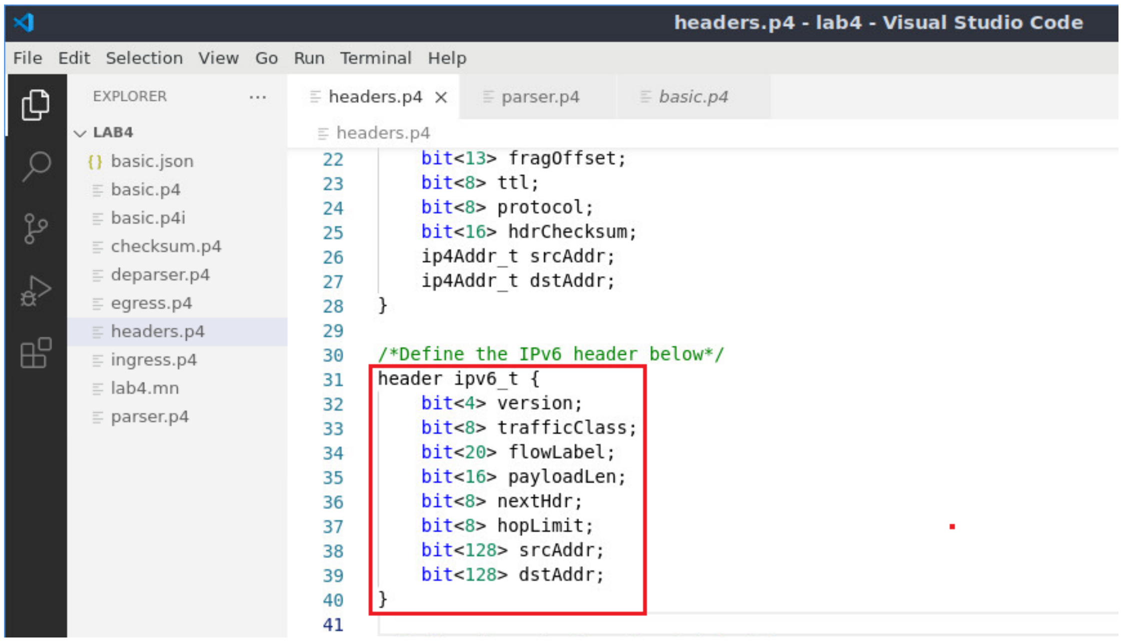
```
/*Define the data type and constant definitions below*/
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;
const bit<16> TYPE_IPV6 = 0x86dd;
```

Figure 36. Adding the IPv6 type definition.

Step 2. Add the IPv6 header definition as shown below.

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

```
        bit<128> srcAddr;
        bit<128> dstAddr;
}
```

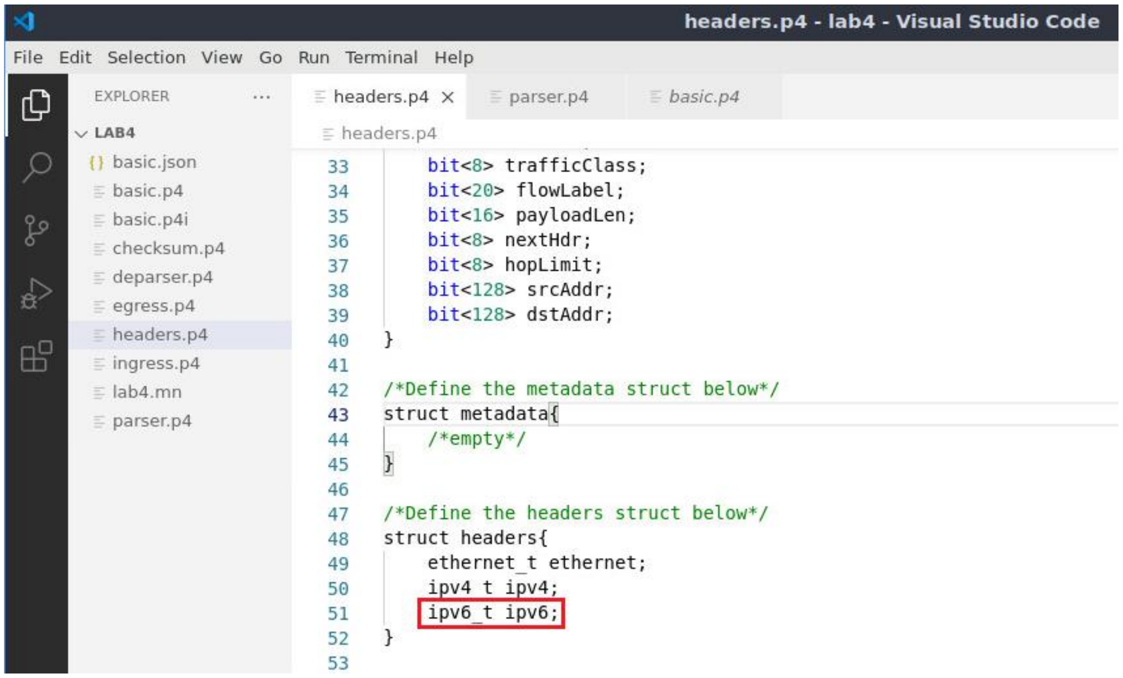


```
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/
header ipv6_t {
    bit<4> version;
    bit<8> trafficClass;
    bit<20> flowLabel;
    bit<16> payloadLen;
    bit<8> nextHdr;
    bit<8> hopLimit;
    bit<128> srcAddr;
    bit<128> dstAddr;
}
```

Figure 37. Adding the IPv6 header definition.

Step 3. Append the IPv6 header to the header's data structure.

```
ipv6_t ipv6;
```

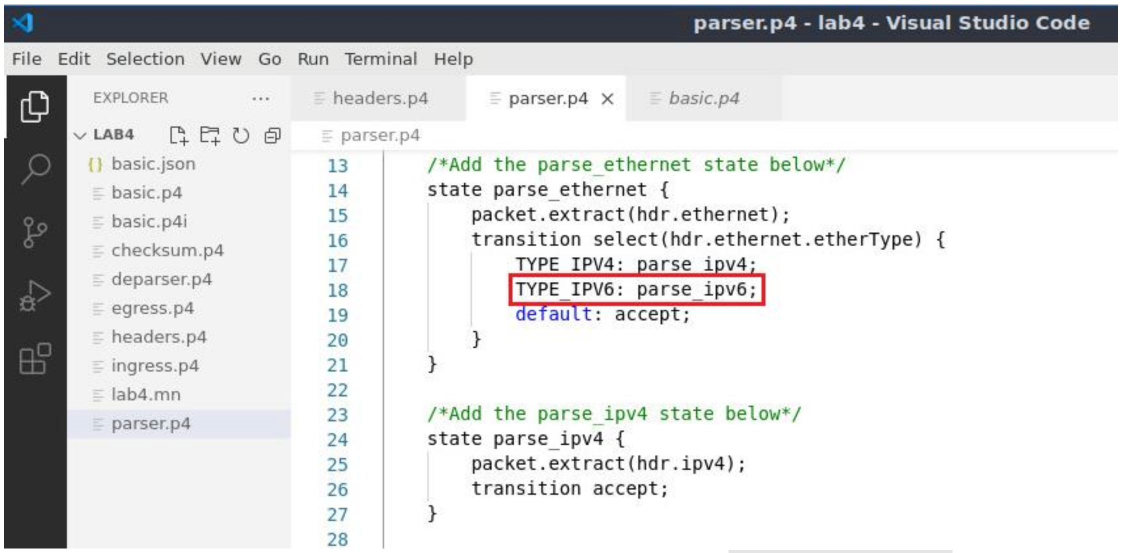


```
    bit<8> trafficClass;
    bit<20> flowLabel;
    bit<16> payloadLen;
    bit<8> nextHdr;
    bit<8> hopLimit;
    bit<128> srcAddr;
    bit<128> dstAddr;
}

/*Define the metadata struct below*/
struct metadata{
    /*empty*/
}

/*Define the headers struct below*/
struct headers{
    ethernet_t ethernet;
    ipv4_t ipv4;
    ipv6_t ipv6;
}
```

Figure 38. Adding IPv6 type to the header data structure.

Step 4. Go to the *parser.p4* file and add the following line to the `parse_ethernet` state.

```
TYPE_IPV6: parse_ipv6;
```

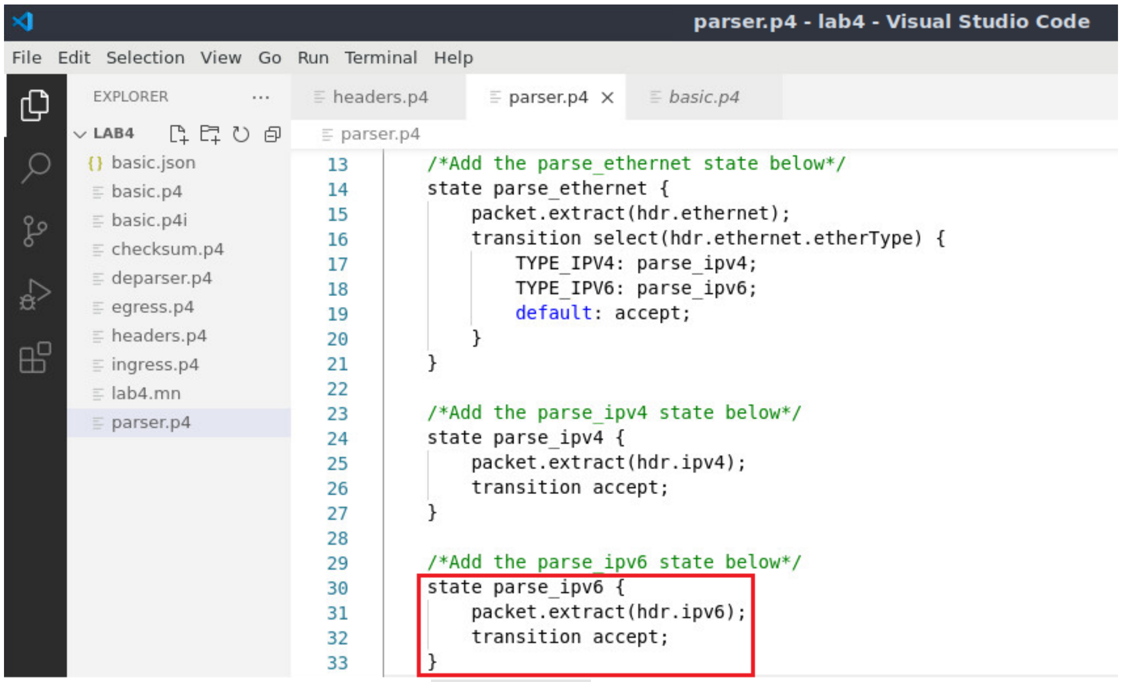


```
/*Add the parse_ethernet state below*/
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        TYPE_IPV6: parse_ipv6;
        default: accept;
    }
}

/*Add the parse_ipv4 state below*/
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```

Figure 39. Including the IPv6 state transition into the `parse_ethernet` state.

Step 5. Add the `parse_ipv6` state inside the parser by inserting the following code.

```
state parse_ipv6 {
        packet.extract(hdr.ipv6);
        transition accept;
}
```



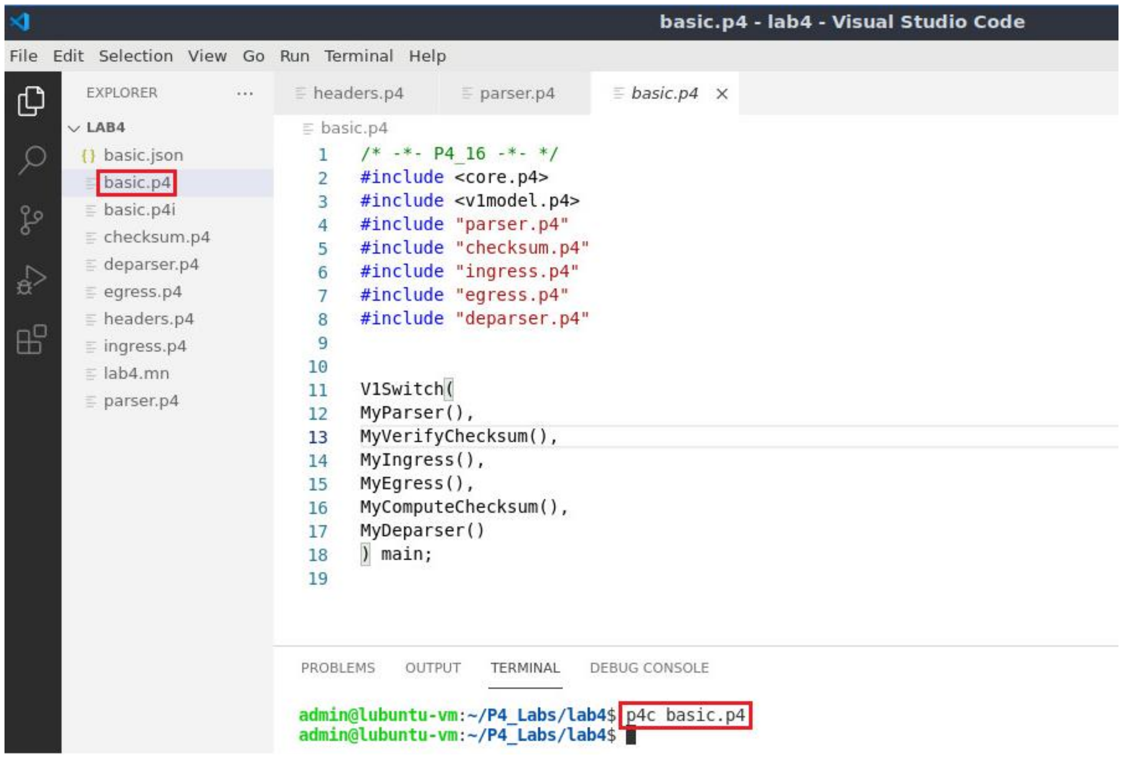
```
/*Add the parse_ethernet state below*/
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        TYPE_IPV6: parse_ipv6;
        default: accept;
    }
}

/*Add the parse_ipv4 state below*/
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}

/*Add the parse_ipv6 state below*/
state parse_ipv6 {
    packet.extract(hdr.ipv6);
    transition accept;
}
```

Figure 40. Adding `parse_ipv6` state to the *parser.p4* file.

Step 6. Save the changes by pressing `Ctrl+s`.

Step 7. Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```



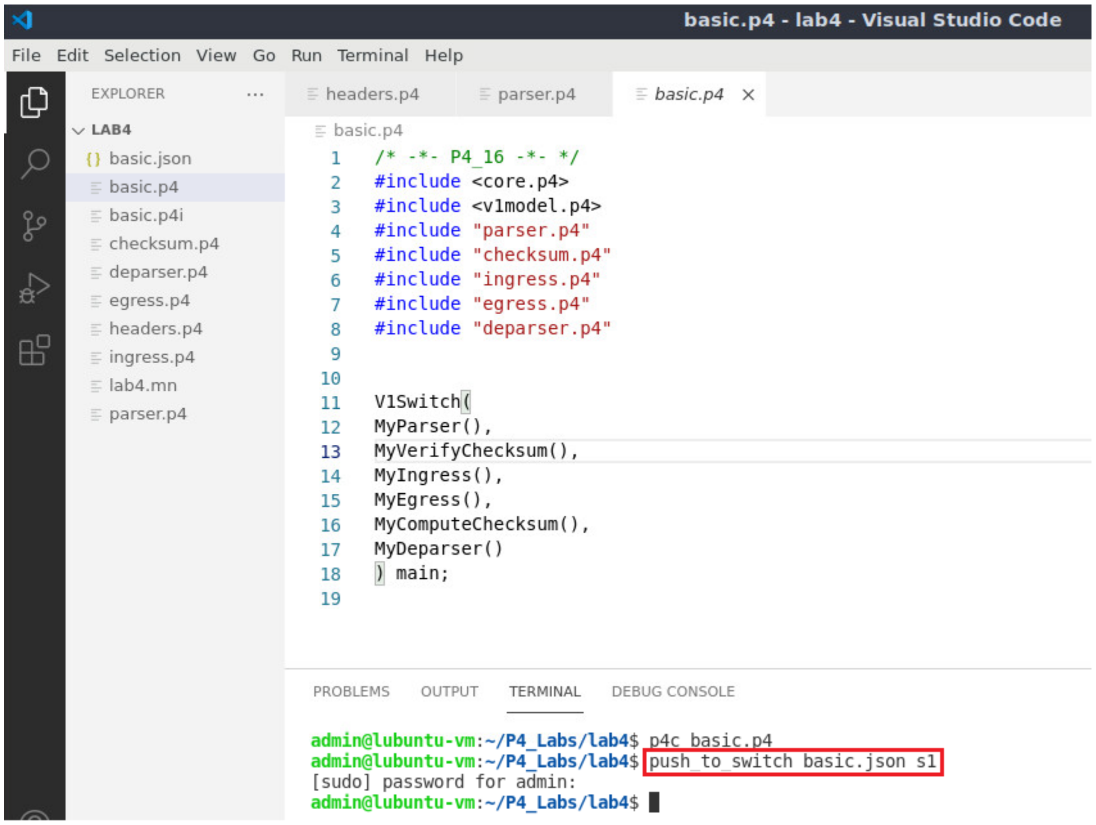
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 41. Compiling the P4 program.

Step 8. Type the command below in the terminal panel to push the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the p4c compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```


basic.p4 - lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB4
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab4.mn
parser.p4
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

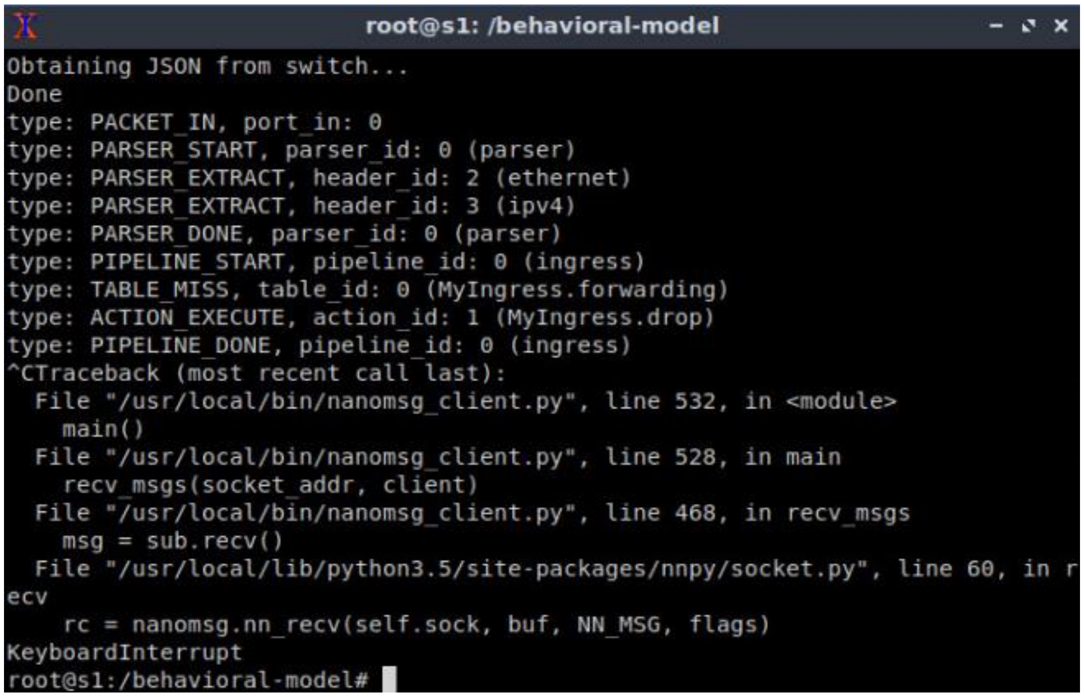
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 42. Pushing the P4 program to switch s1.

9 Testing and verifying the augmented P4 program

Step 1. In switch s1 terminal, press `Ctrl + c` to return to the CLI. The figure below shows the output after executing the command.

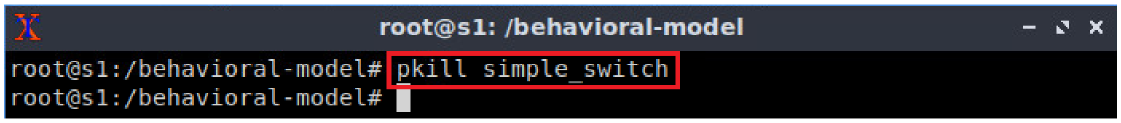


```
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_MISS, table_id: 0 (MyIngress.forwarding)
type: ACTION_EXECUTE, action_id: 1 (MyIngress.drop)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
^CTraceback (most recent call last):
  File "/usr/local/bin/nanomsg_client.py", line 532, in <module>
    main()
  File "/usr/local/bin/nanomsg_client.py", line 528, in main
    recv_msgs(socket_addr, client)
  File "/usr/local/bin/nanomsg_client.py", line 468, in recv_msgs
    msg = sub.recv()
  File "/usr/local/lib/python3.5/site-packages/nnpy/socket.py", line 60, in r
ecv
    rc = nanomsg.nn_recv(self.sock, buf, NN_MSG, flags)
KeyboardInterrupt
root@s1:/behavioral-model#
```

Figure 43. Returning to the CLI.

Step 2. Type the command below in the terminal of switch s1 to stop the running daemon.

```
pkill simple_switch
```

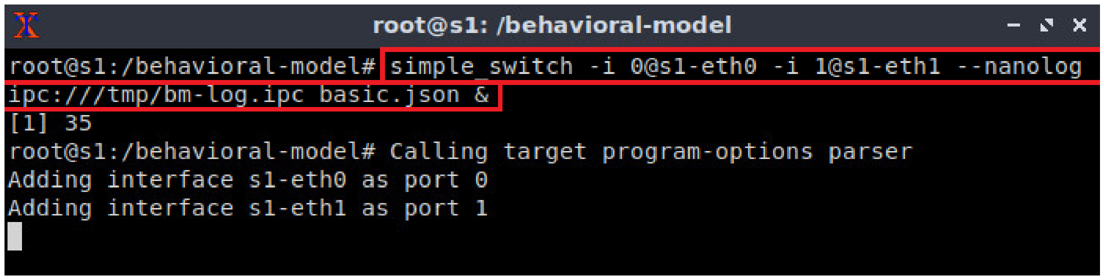


```
root@s1:/behavioral-model# pkill simple_switch
root@s1:/behavioral-model# 
```

Figure 44. Ending switch s1 P4 process.

Step 3. Type the command below in the terminal of the switch s1 to start the daemon with the new P4 program.

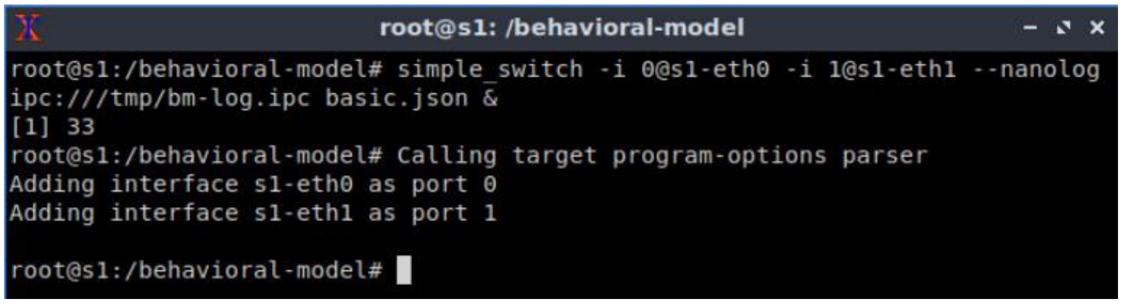
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog ipc:///tmp/bm-log.ipc
basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 35
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 45. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 4. In switch s1 terminal, press *Enter* to return the CLI.

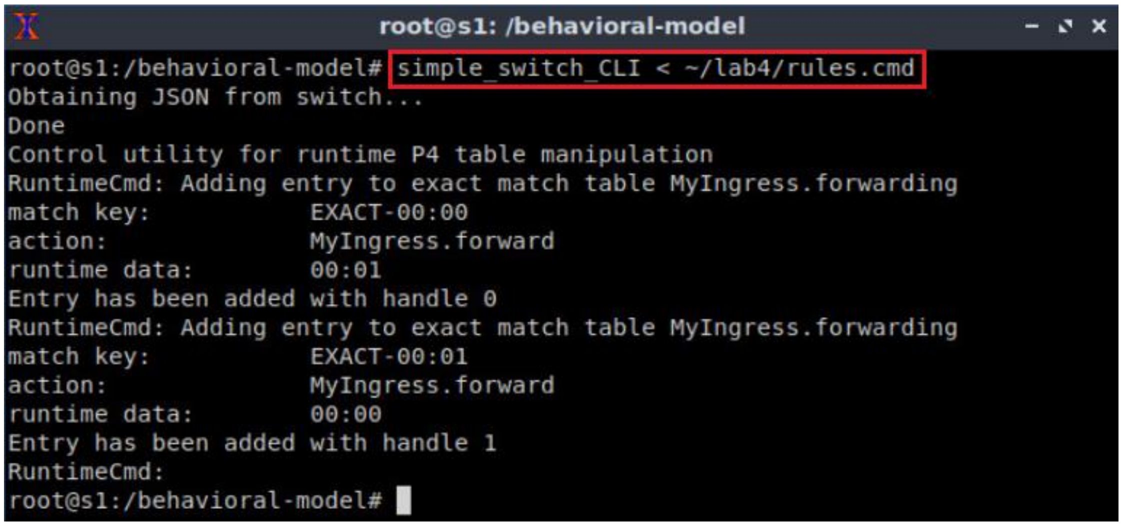


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 46. Returning to switch s1 CLI.

Step 5. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab4/rules.cmd
```

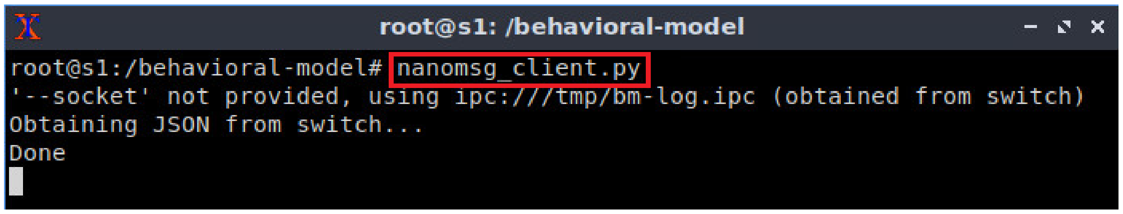


```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab4/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 47. Populating the forwarding table into switch s1.

Step 6. Type the following command to display the switch logs.

```
nanomsg_client.py
```

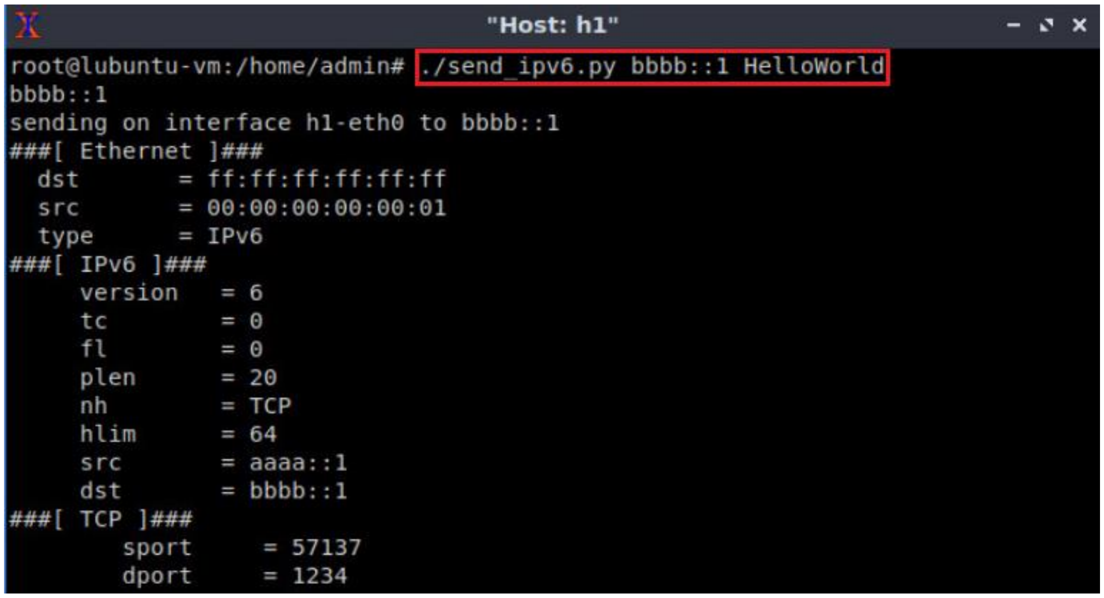


```
root@s1: /behavioral-model
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 48. Inspecting the logs in switch s1.

Step 7. On host h1's terminal, type the following command to send an IPv6 packet to host h2. Note that `bbbb::1` is IPv6 address of host h2.

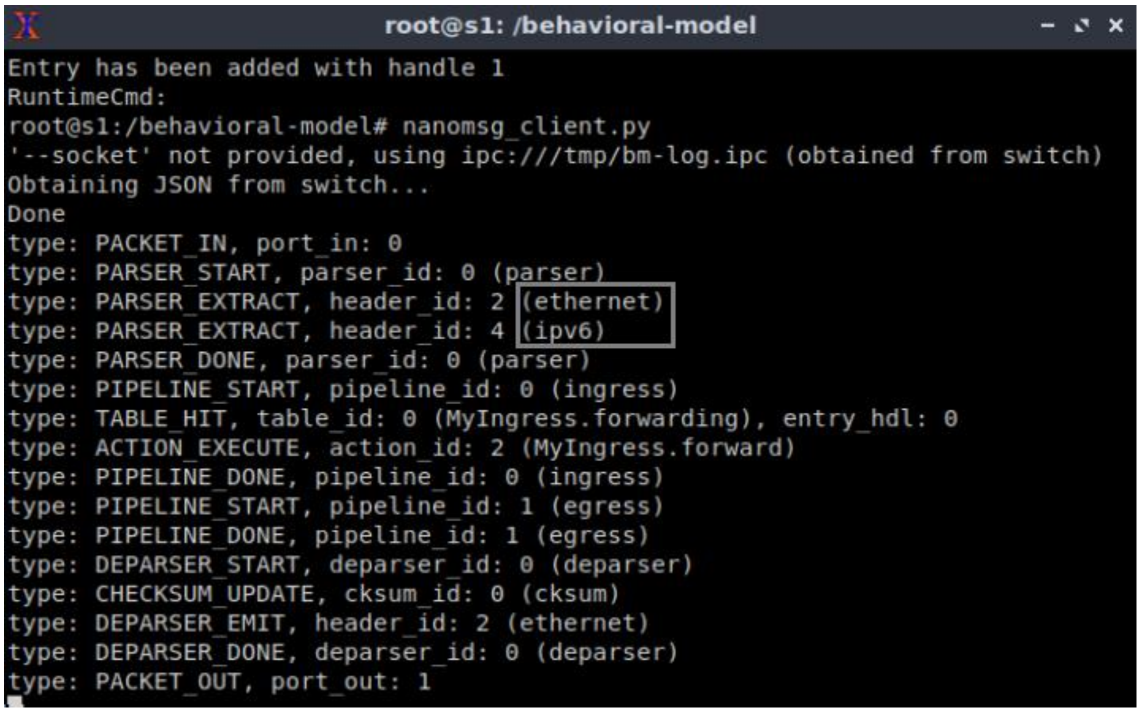
```
./send_ipv6.py bbbb::1 HelloWorld
```



```
"Host: h1"
root@lubuntu-vm:/home/admin# ./send_ipv6.py bbbb::1 HelloWorld
bbbb::1
sending on interface h1-eth0 to bbbb::1
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv6
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 0
     plen      = 20
     nh        = TCP
     hlim      = 64
     src       = aaaa::1
     dst       = bbbb::1
###[ TCP ]###
        sport     = 57137
        dport     = 1234
```

Figure 49. Sending an IPv6 test packet from host h1 to host h2.

Step 8. Go back to switch s1 and inspect the logs.



```
root@s1: /behavioral-model
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 4 (ipv6)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 50. Inspecting the logs in switch s1.

The figure above shows that the Ethernet and IPv6 header are extracted.

This concludes lab 4. Stop the emulation and then exit out of MiniEdit.

References

1. C. Cascaval, D. Daly. "P4 Architectures." [Online]. Available: https://tinyurl.com/3zk8vs6a.
2. “p4c core.p4”. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/core.p4.
3. “p4c v1model.p4”. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.
4. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.