1 Project Description

In this project, I helped develop an Integrated Development Environment (IDE) for the network programming language, P4 [1]. Specifically, I helped develop a plugin for Microsoft VSCode that includes: (i) a syntactic analyzer, (ii) a semantic analyzer for basic errors and usage (e.g., type checker and resource estimator), and (iii) functionality to protect proprietary information.

1.1 Background

Over the last several years, networks have become increasingly programmable. This innovation has the potential to completely change the way companies and customers purchase, customize, and use network technology.

The ability to program switches has numerous benefits. It allows users to tailor the network device to their particular needs. It allows changes to be made at the speed of software development, rather than hardware development. And, it allows for upgrades and bug fixes completely in software. While the concept of network programming is extremely advantageous in theory, since it is a relatively new concept, there are very few tools available that deal with the development process. Additionally, networking programming languages, such as P4 [1], are more restricted than other languages, which make it more difficult to work with.

For example, a vendor like Cisco might develop a program to run their switch titled switch.p4, and needs to share this program with their customers without revealing their implementation. Ideally, customers would be able to change a program through their own separate file, user.p4, without altering anything in Cisco’s program (to avoid ”breaking” Cisco’s code). However, unlike C, P4 does not allow for binary distributions, nor does P4 have a module system. Because of this, we propose to develop an IDE that “hides” proprietary information while allowing for customization.
1.1.1 Personal Focus

This project will create a tool which network operators can use to assist with programming these switches while still preserving some of the proprietary features that the switches hold from the manufacturer and ensuring that these features cannot be compromised in any way.

One way to achieve this goal is to have a certain set of files hosted on the company’s end, including a generic framework (see switch.p4 in the diagram below) and a compiler. The client receives this information and makes it easier for the programmer to add whatever code necessary. The client then sends this combined information back to the server, where it combines the two parts and compiles the program.

The server side will combine user.p4 and switch.p4 by first exposing “hooks” where users can add their extensions to the switch.p4 program. The client will then communicate with the server in order to learn where exactly these “hooks” are, and reveal them to the user so that he can then edit the user.p4 program properly.

The diagram below offers a visual representation of the process:

![Diagram](image)

2 Steps Taken

2.1 Lexical and Syntax Analysis.

This step involved writing and modifying a grammar using Extended Backus-Naur form and used the ANTLR tool to create a parse tree. This was implemented for use in the client side of the environment, allowing the IDE to check for syntactic errors and perform some basic semantic checks without communicating to the server.

2.2 Semantic Analysis

I also implemented simple type checking features into the plugin. Concepts used include type analysis and lexical scoping.
For the purposes of this project, I approached implementing type analysis by using and organizing multiple data structures/concepts in order to quickly and efficiently handle a variety of different scenarios while writing clear and efficient code.

Currently, I have 3 different structures storing information about the program, and they all communicate with one another or work independently, depending on the algorithm that I use.

The first structure is the symbol table, which is standard for all compiler construction. The symbol table stores the identifier of each declaration and the entire node of the parse tree associated with that identifier. Here is a representation using P4 code.

```p4
const bit<32> first = 15;

action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
  standard_metadata.egress_spec = port;
  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
  hdr.ethernet.dstAddr = dstAddr;
  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}

action ipv4_forward2(macAddr_t dstAddr, acAddr_t dstAddr2, egressSpec_t port) {}
```

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>const bit&lt;32&gt; first = 15;</td>
</tr>
<tr>
<td>ipv4_forward2</td>
<td>action ipv4_forward2(macAddr_t dstAddr, acAddr_t...)</td>
</tr>
<tr>
<td>ipv4_lpm</td>
<td>table ipv4_lpm {key = {hdr.ipv4.dstAddr: lpm;}}</td>
</tr>
</tbody>
</table>

The second structure is the type map, which I implemented in order to store identifiers along with their sizes and types. Keeping a separate map which stores the types and sizes eliminates the need to parse further through data stored in the symbol table, making it easier to manage and allowing for more efficient code. Here is a representation adding the following type definitions.

```p4
typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
typedef bit<16> acAddr_t;
```
The third table is a parameter table, which stores the identifier of a function or action along with the parameters it expects, in order. Here is a representation.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipv4_forward</td>
<td>macAddr_t dstAddr, egressSpec_t port</td>
</tr>
<tr>
<td>ipv4_forward2</td>
<td>macAddr_t dstAddr, acAddr_t dstAddr2, egressSpec_t port</td>
</tr>
</tbody>
</table>

Organizing the input into separate tables makes the process of type checking easier. The goal of type checking is to ensure that variables are used correctly and in the correct context. For example, the following code:

```c
function foo()
{
    bool x = true;
    int y = "string";
    return x+y;
}
```

should generate an error from the compiler, since the type of 'x' and the type of 'y' do not match. Additionally, the example:

```c
int s = "string";
```

should return an error as well. Here is a basic algorithm that describes how to solve this problem.

Example type checking function:

```c
lhsType = typeMap.getType(lhs);
rhsType = typeMap.getType(rhs);
if lhsType == rhsType then
    return;
else
    alert(error);
end
```
This algorithm allows the plugin to ensure that the user is assigning the correct type to the expression during the use of a variable.

**Challenges.** One challenge that comes with this algorithm is when multiple expressions are appended together, such as the example

\[ x = x + (y + 5) + z + -2 \]

In this case, it is necessary to split each expression into separate parts and evaluate each part as its own independent use. Normally, this would be a difficult task to complete, as the algorithm would have to take into account many different variables/types/expressions in order to successfully couple accurate left-hand and right-hand sides, but thankfully, ANTLR takes care of each expression by storing them as separate nodes within the parse tree. This can be achieved by using the methods autogenerated by the ANTLR tool.

Another challenge that presents itself when type checking is dealing with a function declaration. Take this example of a P4 function:

```p4
bit<32> max(in bit<32> left, in bit<32> right) {
  if (left > right)
    return left;
  return right;
}
```

When dealing with function declarations, the algorithm described above is useless. In this case, the use of the variable is different than a simple expression; the variable is being used as an argument of a function. In this case, we must compare the type each argument in the function to its corresponding type in the type map, and if they match, we allow the function to continue. Here is a sample algorithm:
Initialize empty data structure (queue for this example);
//getting types of each parameter;
for each parameter in function
    if the parameter is not in the typeMap then
        return error;
    else
        var type = typeMap.getType(parameter);
        //store the type in data structure;
        queue.push(type);
    end
//checking types with function types in symbolTable
symbolTable.get(functionName);
for each parameter in function
    if parameterType != queue.shift() then
        return error
end

2.3 Resource Estimation

One of the biggest challenges for P4 developers is managing resource consumption. Adding relatively minor changes to an existing program can alter the table dependency graph and dramatically impact how a data plane program is laid out in memory. In this project, I helped implement a feature in P4 Weaver to help developers with the resource allocation challenge by giving resource estimates via static analysis of the extension code.

Example resource estimation Function:

let tableSize = Get size value from table;
while there are actions in the table do
    if the action exists in actionTable then
        for each parameter stored in the actionTable
            storer.size += symTable.get(parameter);
    end
    if storer.size is greater than the current maximum size then
        currentMaximum = storer.size;
        storer.size = 0;
    end
end
return currentMaximum * tableSize;

This algorithm references an action table, which is the same table as the parameter
table mentioned during the semantic analysis. This algorithm sums the size of each action, then multiplies it by the total size of the table and reports that value to the user.

**Challenges.** The main challenge with this approach deals with the possibility of multiple actions being performed in the same table. In the following example

```c
control MyIngress() {
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            ipv4_forward2;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

the table defines 2 separate actions. Using the algorithm above, the program will search the tables for the corresponding sizes of each parameter of the action methods.

According to the methods

```c
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    ...
}
```

```c
action ipv4_forward2(macAddr_t dstAddr, acAddr_t dstAddr2, egressSpec_t port) {
    ...
}
```

The following values are selected in the type map:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>egressSpec_t</td>
<td>9</td>
<td>typedef</td>
</tr>
<tr>
<td>macAddr_t</td>
<td>48</td>
<td>typedef</td>
</tr>
<tr>
<td>ip4Addr_t</td>
<td>32</td>
<td>typedef</td>
</tr>
<tr>
<td>acAddr_t</td>
<td>16</td>
<td>typedef</td>
</tr>
</tbody>
</table>
Making the size of the first action $9+48 = 57$, while the size of the second is $9+48+16 = 73$. At first, my code would add the two independent sums together and return that result times the size of the table, but that way is the incorrect approach. The reason for resource estimation is to ensure that the developer understand the maximum amount of resources needed to complete a certain task at one time, not the total amount of resources used over the entire course of the program. For this reason, as shown in the algorithm written above, we set a maximum value of resource used at a given time, and only update that max if it is surpassed. It is the maximum value that we multiply by the total size of the table and then return to the user.

### 3 Future Work

In the future, I plan to tighten up some of my code and make it more efficient, as well as making sure that all possible test cases are covered. When dealing with user input in compilers, I learned that there are a plethora of different ways users can write code that produces identical results, so I need to make sure that all of those ways are accounted for.

Additionally, I will work on the client side of the plugin to make it easier for the resources to be seen for each individual table. I plan on implementing a command where the user can select whichever plugin he/she desires, and a pop-up will emerge displaying the amount of resources for that particular table, as well as the maximum for the entire program.

### 4 Conclusion

Throughout the semester, I helped work on P4 Weaver, a development environment to enable incremental programming on P4-based switches. The design of P4 Weaver is inspired by prior work on Aspect-Oriented programming. But, in contrast to traditional AOP, P4 Weaver adopts a principled and controlled approach to code isolation, by assigning clearly distinct roles, vendor and customer, with a proper separation of concerns. Moreover, P4 Weaver provides controlled visibility via a client-server architecture that prevents physical access to the vendor base-program. Using three real-world inspired case studies, we have demonstrated how P4 Weaver allows users to address their use cases in the most suitable way by adding their own custom features, while also leveraging typical switching and routing functions. Overall, P4 Weaver enables the incremental programmer to leverage the base code without needing to cope with its complexity and while keeping the base code confidential.