# Perdix: The iptables Query Language

Justin Prosco
jp2415@columbia.edu

Angelika Zavou
az2172@columbia.edu

Orr Bibring
ob2135@columbia.edu

Bing Wu
bw2236@columbia.edu

December 18, 2007

# Contents

# 1 Introduction

Perdix is a high-level scripting language for the analysis of netfilter.org `iptables` firewall logs. The language is designed to make parsing of logs more intuitive and easily scripted without the use of shell utilities such as `grep`, `uniq`, and `sort`, which are typically called in other scripting languages such as `bash`.

## 1.1 Name

Perdix is a language used to efficiently cut through large `iptables` log files. The saw is the tool that is traditionally used to cut through logs. In Greek mythology, Perdix invented the saw, which was modeled after the backbone of a fish.

## 1.2 Motiviation

Perdix was designed for the use of writing compact scripts to parse through Linux `iptables` firewall logs. These logs can get very large if they are not rotated, and sorting through them often requires the use of several different Unix utilities.

Microsoft provides a tool named Log Parser that provides similar functionality to Perdix by allowing the programmer to use standard SQL syntax to parse through security logs. The product only works on Windows, however, and can only be used to parse Windows related logs. Perdix attempts to provide a language with similar functionality to this application on the Unix platform.

## 1.3 Goal

The goal of Perdix is to provide the programmer with a powerful scripting environment for the parsing of netfilter log files. It also provides the ability for users to use a simplified query syntax to parse large log files. While Perdix does not allow the programmer to use standard SQL query expressions, it contains significant relational and logical query capabilities that extend the functionality of the traditional Unix command line.

The output of the Perdix compiler produces Java code that is compatible with JRE 1.5 and higher. Since Java is platform independent, the output files can be ported to any system capable of running a Java Virtual Machine (JVM).

## 1.4 Language Features

The data types in Perdix have been optimized so that they match the type of network security information typically found in these log files. The data types correspond with the content typically found in these logs, rather than with generic data types.

Code in Perdix is written using English statements. Whitespace is irrelevant to the final executable and is stripped by the compiler. Each statement is completed by a semicolon (;), though function declarations contain an implicit semicolon delineated by the right curly brace (}).

The data types in Perdix have been customized for the field names found in `iptables`. There are specific data types for both network interface MAC addresses and IPv4 addresses. A sample security log entry from the `iptables` firewall is found below:

```
Feb  1 00:00:02 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=192.150.249.87 DST=11.11.11.84 LEN=40 TOS=0x00 PREC=0x00
TTL=110 ID=12973 PROTO=TCP SPT=220 DPT=6129 WINDOW=16384 RES=0x00 SYN URGP=0
```

# 2 Language Tutorial

This language tutorial is intended for a first-time Perdix programmer. It introduces the basic concepts of the language and the structure of Perdix programs.

## 2.1 A Simple Perdix Program

Every Perdix program has the same basic structure. User-defined functions are defined at the top of a file, followed by the main body of the program which is declared using the `start()` function.

For this example we assume that we have an iptables log file on the local filesystem at /var/log/iptables.log. This file contains the following four entries:

```
Feb  1 00:00:02 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=192.150.249.87 DST=11.11.11.84 LEN=40 TOS=0x00 PREC=0x00
TTL=110 ID=12973 PROTO=TCP SPT=220 DPT=6129 WINDOW=16384 RES=0x00 SYN URGP=0

Feb  1 00:00:02 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=24.17.237.70 DST=11.11.11.95 LEN=40 TOS=0x00 PREC=0x00
TTL=113 ID=27095 PROTO=TCP SPT=220 DPT=6129 WINDOW=16384 RES=0x00 SYN URGP=0

Feb  1 00:00:07 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=192.150.249.87 DST=11.11.11.85 LEN=40 TOS=0x00 PREC=0x00
TTL=110 ID=13801 PROTO=TCP SPT=220 DPT=6129 WINDOW=16384 RES=0x00 SYN URGP=0

Feb  1 00:00:17 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=192.150.249.87 DST=11.11.11.87 LEN=40 TOS=0x00 PREC=0x00
TTL=110 ID=15432 PROTO=TCP SPT=220 DPT=6129 WINDOW=16384 RES=0x00 SYN URGP=0
```

To analyze these logs, we would like to write a Perdix program to find the entries that have a source IP address of 24.17.237.70 or an ID of 15432.

Since we are not defining any functions, we will begin the source with the `start()` function. From there, we declare a file type variable and initialize it to the log file. We then open the log file using the `open()` command. We must do the same for the file where the results of the query will be stored. Notice that the logfile is opened with the "r" option for reading and the results file is opened with the "w" option for writing.

The conditional statement is done in the `cond()` statement, where we will define the log file to work on in the first argument. Since we have a compound logical statement, we use the logical OR operator in the `cond()` function. The condition statement filters the results output by the function. If there is no condition specified, the function works on every line in the file.

Inside the `cond()` body are two statements. The `results()` statement contains the keyword screen and two iptables keywords. This will print all of columns SRC and ID from the matches of the conditional statement. Since screen is specified as the file type, the results will be printed to standard out. The second `results()` statement has two arguments, the file variable "resfile" and the iptables keyword SRC. This will print all of the source ip addresses matching the query to a CSV file "results.txt."

Following the `cond()` statement is the closure of both the files that were previously opened.

The full code listing for this program is found below:

```
function start()
{
    file logfile = "/var/log/iptables.log";
    file resfile = "results.txt";

    open(logfile,"r");       # open file for reading
    open(results,"w");       # open file for writing

    cond(logfile : SRC==@24.17.237.70 || ID==15432)
    {
        # print full lines to the screen
        results(screen, SRC, ID);

        # print only destination ip addresses to file
        results(resfile,SRC);
    }

    close(results);
    close(logfile);
}
```

## 2.2 Compiling the Source

To compile the source, run the following command:

```
$ perdix myfirstperdix.per
```

If there are errors in the code, they will be displayed to the screen. If not, a Java class file (myfirstperdix.java) will be created in the current working directory.

## 2.3 Running the Compiled Output

Perdix generates a Java class file. To run the query against the data, this class file first needs to be compiled using a Java compiler. To do this using, type the following:

```
$ javac -o myfirstperdix myfirstperdix.java
```

When Java compilation has finished, the program can now be executed on the Java Virtual Machine (JVM) by typing:

```
$ java myfirstperdix
```

# 3 Perdix Language Reference Manual

## 3.1 Anatomy of a Perdix Program

Perdix programs have the same basic structure. User-defined functions are declared at the top of a file. Variable declaration and initialization is done at the top of a function and is followed by a sequence of additional statements. Every program must also contain the `start()` function with the body of the main program contained within its curly braces.

Queries are executed using the `cond()` function and options to return the results of these queries are enclosed within the curly braces surrounding the conditional body. Files must be opened before they can be passed to the `cond()` statement and should be closed before the end of the program.

An example Perdix program skeleton is found below. Although conditional statements are not required, it is the core feature of the Perdix language.

```
function declarations { }

function start()
{
    variable declarations and initialization
    expressions
    conditional statements { conditional body }
    expressions
}
```

## 3.2 Lexical Conventions

### 3.2.1 Tokens

Tokens area divided into the following classes:

- Constants
- Identifiers
- Keywords
- Operators
- Punctuation

Any non-printable character such as a space, tab, new line, and form feed are considered whitespace. Whitespace is discarded by the compiler, but is available to the programmer to make a program more human readable. Whitespace is not used to dictate the scope of variables or program flow, but is used to separate tokens.

### 3.2.2 Constants

#### Integer Constants

An integer constant is a sequence of digits. All integers are positive base 10 numbers and do not need to be enclosed in double quotation marks when being defined.

#### String Literals

String literals are arbitrary length sequences of characters enclosed in double quotation marks. If a string literal contains double quotation marks, then they are written twice as shown below:

```
string test = "The ""double quotes"" are also part of the string";
```

### 3.2.3 Identifiers

Identifiers refer to user-defined variable names. An identifier must begin with a letter or underscore and can contain a series of letters, underscores, or numbers as long as the name of the variable does not match one of the reserved words. Identifiers may contain either uppercase, lowercase, or mixed case letters. All identifiers in Perdix are case sensitive.

### 3.2.4 Keywords

Keywords are reserved words within the Perdix language that cannot be used as identifier names.

#### Perdix Keywords

The following are keywords in the Perdix language:

| | | | |
|---|---|---|---|
| **cond** | **int** | **file** | **string** |
| **start** | **open** | **print** | **close** |
| **array** | **function** | **write** | **ip4address** |
| **mac** | **unique** | **results** | |

#### iptables Keywords

In addition, the following iptables keywords are available in Perdix. Each keyword also has a type associated with it. This indicates the type of data that must appear as an rvalue in a relational statement using this keyword. Although there are no specific boolean values in Perdix, the keywords with a boolean integer type indicate that this keyword can be related to 0 (does not appear in current line of file) or 1 (appears in current line of file). Any value other than 0 is assumed to be 1 by the compiler.

- **DATE**: the timestamp of the packet (string)
- **IN**: the interface the packet came in on (string)
- **PHYSIN**: the physical interface the packet came in on (string)

- **OUT**: the interface the packet was sent (string)
- **PHYSOUT**: the physical interface the packet was sent on (string)
- **MAC**: the MAC address of the adapter (mac)
- **SRC**: the source IP address of the packet (ip4address)
- **DST**: the destination IP address of the packet (ip4address)
- **LEN**: the length of the IP packet in bytes (int)
- **TOS**: the type of service - type field (string)
- **PREC**: the type of service - precedence field (string)
- **TTL**: the packet time to live (integer)
- **ID**: unique IP for IP datagram (integer)
- **CE**: congestion experienced flag (boolean integer)
- **SPT**: the source port of the packet (integer)
- **DPT**: the destination port of the packet (integer)
- **SEQ**: sequence number (integer)
- **WINDOW**: the window size of the packet (integer)
- **RES**: reserved bits (string)
- **SYN**: TCP SYN flag (boolean integer)
- **ACK**: Acknowledgment flag (boolean integer)
- **PSH**: Push flag (boolean integer)
- **RST**: Reset flag (boolean integer)
- **FIN**: FIN flag (boolean integer)
- **URGP**: Urgent Pointer flag (boolean integer)

#### 3.2.5 Punctuation

There is a limited amount of punctuation defined in the Perdix grammar:

- Semicolon ( ; ) : signifies the end of a Perdix statement
- Comma ( , ): used to separate a list of arguments passed to functions
- Colon ( : ): used in the cond function to separate the file argument from the conditional statement
- Double Quotation Marks ( “ ” ): indicate a string literal
- Braces ( { } ) : define the body of a Perdix function
- Parentheses ( ( ) ): used to contain the arguments passed to functions. Parentheses can also be used to enforce precedence in conditional statements
- Pound ( # ): signifies the beginning of a programmer comment
- At ( @ ): used to assign an ip4address value

#### 3.2.6 Comments

Comments are a single line beginning with the `#` character. Anything following a `#` character up to the newline character is assumed to be a comment and will be discarded by the compiler. Multiline comments are not allowed, as each line must begin with the `#` character. An example of this is shown below:

```
# This is a comment
# This is the second line of the comment
int f=5;  # Comment at the end of a line
```

### 3.3 Data Types

#### 3.3.1 int

The int type is used to declare integer variables. Due to the nature of iptables log files, there is no need for negative values, as there should never be a negative integer value in an iptables log file. Constant values assigned to variable of this type do not need to be enclosed in double quotation marks.

#### 3.3.2 string

The string data type is used to define string literals. Variables of this type can be assigned a value of any arbitrary length character string as long as it is enclosed in double quotation marks.

#### 3.3.3 file

The file data type is a special type of string that is used to represent file variables. The value of a file variable should be an operating system-specific path to a file, including the file name, enclosed in double quotation marks. Therefore, file data types are defined with string literal values.

#### 3.3.4 array

The array data type contains a collection of other types. Arrays are defined as containing a single fundamental data type. For example, if an array is defined as an integer array, it can only hold elements that are of the integer type.

#### 3.3.5 ip4address

The ip4address type defines an IPv4 address in dotted form. This data type is used to declare single IP address types. An IPv4 address consists of four octets and has a value range of 0.0.0.0 - 255.255.255.255. Every value assigned to an ip4address must start with the @ character. For example, @192.168.1.1 is a valid ip4address value in Perdix.

#### 3.3.6 mac

The mac type is used to define media access control (MAC) addresses. This is the hardware address of a network interface card. MAC addresses are 48-bit numbers consisting of 6 octets. Each octet is represented in hexadecimal (base-16) and is separated from other octets by a colon. An example of a MAC address is 00:00:00:aa:aa:aa.

### 3.4 Functions

Perdix provides the ability for users to define their own functions. User-defined functions must appear above the `start()` function in the main file of a Perdix program. Perdix functions can not return values when they complete execution, so they simply provide the programmer with the convenience of reusing code. A common use of functions is to use the same query conditions on multiple input log files.

#### 3.4.1 Function Declarations

Functions are declared by using the keyword "function" followed by an identifier and an optional, comma-separated list of parameters and their data types enclosed in parentheses. An example of the declaration of a function named myfunction that takes two parameters is:

```
function myfunction(int integer1, string string1) { }
```

#### 3.4.2 Function Calls

A function may be called anywhere within another function. This is accomplished by calling the function by its identifier and passing identifiers or values as the parameters. Using the example above, the function myfunction can be called in several different ways, including the following:

```
myfunction(80, "ACCEPT");
myfunction(var1,var2);   # var1 is an int, var2 is a string
myfunction(var1, "string");
```

### 3.5 Expressions

Expressions are combinations of constants, variables, operators and punctuation that are evaluated according to the particular rules of precedence and associativity defined in Perdix. Common locations for expressions are in the `cond()` statement as well as in the initialization of variables. There are four types of expressions in Perdix, arithmetic expressions, conditional expressions, assignment expressions and logical expressions.

#### 3.5.1 Arithmetic Expressions

Arithmetic expressions can be used in Perdix utilizing the addition and subtraction operators. These expressions can be substituted anywhere an integer value is accepted. A common use of arithmetic expressions is to assign a variable the modified value of another previously defined variable as shown below:

```
int var1 = 200;
int var2 = var1 + 30;
```

### 3.5.2 Conditional Expressions

Conditional expressions are relational expressions that involve the use of the >, <, >=, <=, ==, or != operators and are used inside the `cond()` function. The lvalue of all conditional expressions will be one of the iptables keywords, as a conditional statement implies putting a constraint on the query for a particular field within a log file.

Conditional expressions do not return a value. Rather, the lvalue of the statement is changed at runtime to the value of the field specified on the current input line. For example, the condition `SPT==80` says if the source port on the current line is equal to 80, this line is a conditional match and is included in the results.

If the boolean expression result is true, then the current line of the input log file will be included in the query results. If the statement evaluates to false at run time, the current line of the input log file is not included in the query results.

The relational operators >, <, >=, <= may be used on int, ip4address and mac datatypes. The equality operators == and != may be used on any data type.

Examples of conditional expressions are:

```
DST == @192.168.100.1;
DPT >= 80;
```

### 3.5.3 Assignment Expressions

There are two ways in which an assignment statement can be used, when initializing a variable at declaration and when setting the value of a variable to a new value. The assignment expression uses the equal operator (=). The lvalue is a variable and the rvalue must agree with the type of the variable it is being assigned to.

Although variable declarations with initialization occur at the beginning of a function, assignment statements that are not initial declarations may occur at any place other than within a function.

Examples of assignment expressions are:

```
variable1 = 80;
variable1 = variable2;
```

### 3.5.4 Logical Expressions

Logical expressions are found exclusively within the `cond()` statement and are a sequence of one or more conditions that are tied together by one of the logical operators. To include all conditions, the logical AND operator is used. To include one or more condition, the logical OR operator is used. Finally, to negate all the conditions specified in a query, the NOT operator is used.

An example of a logical expression is:

```
DST == @192.168.100.1 && DPT == 80 || !(OUT == "eth0")
```

## 3.6 Operators

There are four classes of operators available in Perdix: arithmetic, relational, equality and logical.

### 3.6.1 Arithmetic Operators

Perdix contains addition and subtraction arithmetic operators that can be used to combine integers:

- **Addition**: *lvalue + rvalue*
  The result of expressions using this operator will be the sum of the lvalue and rvalue.
- **Subtraction**: *lvalue - rvalue*
  The result of expressions using this operator will be the lvalue minus the rvalue.

### 3.6.2 Relational Operators

Perdix contains the following relational operators that can be used to relate integers, ip4addresses, and mac addresses:

- **Greater Than**: *lvalue > rvalue*
  Evaluates to true when the lvalue is strictly greater than the rvalue.
- **Less Than**: *lvalue < rvalue*
  Evaluates to true when the lvalue is strictly less than the value of the rvalue.
- **Greater Than or Equal**: *lvalue >= rvalue*
  Evaluates to true when the lvalue is greater than or equal to the rvalue.
- **Less Than or Equal**: *lvalue <= rvalue*
  Evaluates to true when the lvalue is less than or equal to the rvalue.

### 3.6.3 Equality Operators

Equality operators can be used with any data type. The following equality operators are defined in Perdix:

- **Equal** : *lvalue == rvalue*
  Evaluates to true when the lvalue equals the rvalue.
- **Not Equal**: *lvalue != rvalue*
  Evaluates to true when the lvalue does not equal the rvalue.

### 3.6.4 Logical Operators

Logical operators are also defined in order to add or subtract results from the returned queries. These operators evaluate to a boolean value of 1 (true) if the constraints specified are matched, and 0 (false) if they are not. The following logical operators are defined:

- **Logical And** ( `&&` ) : *conditional_expression* `&&` *conditional_expression*
  Evaluates to true when both the lvalue and rvalue are true

- **Logical Or** ( || ): *conditional_expression* || *conditional_expression*
  Evaluates to true when either the lvalue or rvalue or both are true
- **Negation** ( ! ): ! *(conditional_expression)*
  This operator negates the rvalue of the statement it precedes. If the statement after the negation operator evaluated to true, then its negation will be false. Likewise, if the statement after the negation operator evaluated to false then the output will be true. It should be noted that the rvalue must be enclosed in parentheses for clarification.

#### 3.6.5 Precedence

All statements in Perdix are evaluated from left to right and are left-associative. Parentheses may be used in logical expressions to force precedence.

The precedence of operators are organized in the table below from highest to lowest.

| Precedence | Operators | Description | Associativity |
|---|---|---|---|
| 1 | ( ) | Parentheses | left to right |
| 2 | ! | Logical Not | left to right |
| 3 | + , - | Addition and Subtraction | left to right |
| 4 | = | Assignment | left to right |
| 5 | > , < , <= , <=, == , != | Relational and Equality Operators | left to right |
| 6 | && , \|\| | Logical And, Or | left to right |

### 3.7 Expressions and Statements

This section describes the statements found in Perdix.

#### 3.7.1 Variable Declaration

Variable declarations are assignment statements that occur at the top of every function body. Variable names must begin with a letter or underscore and can contain a series of letters, underscores, or numbers so long as the name of the variable is not one of the Perdix reserved words.

Variables must be given a data type and also initialized during declaration and cannot be referenced until doing so. Initialization must occur in the same line where the variable is initially declared. The only exception to this rule is the declaration of associative arrays, which do not have to be initialized when declared. The syntax for this is the same as the syntax for an assignment statement, but with an added data type declaration:

```
<data type> <identifier> = <expression>;
```

Variables may not be assigned to another variable if the second variable has not yet been defined with a value. In addition, only one variable can be declared per line. Some examples of variable declaration are:

```
int variable1 = 20;
string variable2 = "literal value";
int variable3 = variable1;
```

### 3.7.2 Array Declaration

```
array <data type> <identifier> [<data type>]*;
```

Array declarations are a special type of variable declaration since they take two keyword values, "array" and a data type that defines the type of information stored in the array. Another distinction from regular variables is that arrays do not have to be initialized at declaration. All arrays in Perdix are fully associative and are capable of being multidimensional.

To initialize the array, a comma separated list of values enclosed between curly braces is included after the assignment operator. Some examples of the various ways to define an array are:

```
array int A[int];
array int B[int][ip4address];
```

### 3.7.3 Assignment Statement

```
 <identifier> = <expression>;
```

Variables are assigned values by using the assignment operator. A variable may be defined as any of the data type values, as long as the data type being assigned matches the type of the identifier.

### 3.7.4 Start

```
function start() { perdix_body }
```

The `start()` statement signifies the start of execution in a Perdix program and should only appear once in any program. The `perdix_body` contains the program body.

### 3.7.5 Open

```
 open(file, file_perm);
```

The `open()` statement opens an existing file. This statement takes two arguments, both of which are required. The first argument is a file identifier, specifying the path of the file to be opened. The file variable should be defined before using it within an open statement. The second argument is a file permission mode. The arguments must be separated by a comma.

#### File Permission Modes

There are two file permission modes in Perdix, "read" and "write." These modes use the values `‘‘r"` and `‘‘w"` respectively. While they are not keywords, the built-in functions that use these modes specifically check whether or not they have been supplied as an argument to the the `open()` function. Only a single permission mode can be supplied to a function at a time.

### 3.7.6 Condition

```
cond(file : conditional_statements) { conditional_body }
```

All query statements in Perdix are defined with the `cond()` statement. This statement takes a file to read from as the first argument and then a query statement as the second argument. Directly following a `cond()` statement is a set of curly braces in which other functions such as `print()`, `write()`, `unique()`, and `count()` may be used in order to define the output. The conditional statement does not return any values without the use of these additional functions.

### 3.7.7 Print

```
print(file_id, variable_list);
```

The `print()` statement allows the contents of variables or constants to be printed to a file or the screen. If the first argument is the keyword `screen`, the variables will be printed to standard output. Otherwise, the values of the variables will be printed to the file specified. The second argument, separated from the first by a required colon, is a comma separated list of variables or constants to be printed. The `print()` function can be called within any function, including `cond()`.

### 3.7.8 Results

```
results(file_id, iptables_keywords);
```

The `results()` function is similar to the `write()` function, though it does not have the ability to write the results to an array. The first argument is a file identifier of where the results matching a `cond()` statement should be written. The second argument(s) are separated from the first by a comma and contains a comma separated list of the iptables keywords that should be included in the output. Any amount of iptables keywords may be specified. This function produces output in comma separated value (CSV) format, so they can easily be imported into a spreadsheet for further analysis and reporting. This function can only appear within the body of a `cond()` function.

### 3.7.9 Write

```
write(array_id, iptables_keywords);
```

The `write()` function takes the iptables keywords specified from the results of a `cond()` statement and stores them to an array. Both the array argument and iptables keywords arguments are required. This function can only appear within the body of a `cond()` function.

### 3.7.10 Unique

```
unique(array_id, iptables_keyword);
```

The `unique()` function is used to filter the results of a `cond()` statement and can be used only in the body of a `cond()`. This function is used to provide a sum of the distinct values for an iptables keyword

within the query results. The `unique()` function takes two arguments, both required. The first argument is an identifier of an array to write the results to. The second argument is the iptables keyword to print the unique values of.

#### 3.7.11 Count

```
count(array_id, iptables_keyword);
```

The `count()` function is used to provide statistical information about information obtained from a `cond()` statement. The first argument is an associative array to write the results to. The second argument is a single iptables keyword to be counted. Multiple calls to `count()` must be used if more than one keyword needs to be counted. This function can only appear within the body of a `cond()` function.

#### 3.7.12 Close

```
close(file_id);
```

The `close()` statement closes a file. This statement takes a single, required argument, the file to close.

### 3.8 Scope

Perdix programs are statically scoped and all variables are local to the function in which they are declared and initialized in. Variables may be passed to other functions as parameters, but they are passed only by value. A copy of the value contained in the variable is used in the function in which it is passed.

Each function declaration has its own scope. The `cond` function is the only built-in function that includes an inline body. This body definition begins a new scope, but inherits the values from the previous scope from which it was called.

# 4 Project Plan

## 4.1 Team Responsibility

Below is the high-level overview of the contributions made by each group member to the project.

| Team Member | Responsibilities |
|---|---|
| Justin Prosco | (Lead) LRM, Final Report, Parser, Java Classes, Java Code Generation |
| Angelika Zavou | (Co-Lead) Final Report, Lexer, Parser, Walker, Java Classes |
| Orr Bibring | Java Classes, Testing |
| Bing Wu | JUnit, Testing |

## 4.2 Java Class Responsibility

The responsibility for classes listed below are based on CVS revision history of classes included in the final version of the Perdix compiler.

- PerdixGrammar.g : Justin, Angelika
- PerdixWalker.g : Angelika
- Node.java : Justin, Angelika
- Stmt.java : Justin, Angelika
- Cond.java : Justin
- FunctionDeclaration.java: Justin, Angelika
- FunctionCall.java : Justin, Angelika, Orr
- Block.java : Justin, Angelika
- Program.java : Justin, Angelika, Orr
- ExprStmt.java : Angelika
- Expr.java : Justin, Angelika
- BooleanConstant.java : Justin, Angelika
- StringConstant.java : Justin, Angelika
- IntConstant.java : Justin, Angelika
- IpConstant.java : Justin, Angelika
- MacConstant.java : Justin, Angelika
- Assign.java : Justin, Angelika
- Not.java : Justin, Angelika
- Variable.java : Justin, Angelika
- VariableDeclaration.java : Justin, Angelika

- Operation.java : Justin, Angelika
- Type.java : Justin, Angelika, Orr
- PerdixArray.java : Justin, Angelika, Orr
- PerdixFunction.java : Justin
- PerdixException.java : Angelika
- SymbolTable.java : Justin, Angelika
- Perdix.java : Justin, Angelika

## 4.3 Coding Conventions

The standard Java coding conventions were followed for Perdix. The coding conventions can be found at http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html (Revised April 20, 1999). The Eclipse built-in tool for code-formatting was extremely helpful in conforming to the indentation rules mentioned in this document.

## 4.4 Development Environment

We used the following tools in the development of Perdix:

- **Development IDE:** Eclipse IDE 3.2
  Eclipse was used as the Java IDE of choice by the group. This is because it has integration with CVS for version control and also contains a plugin for using ANTLR grammars within a project.
- **Development Language:** Sun Java 1.5
  The latest version of Eclipse supports many versions of Java. The default version that is supported is Sun Java 1.5, which is what we decided to use as the JVM for Perdix as well. The version of ANLTR used seems to generate some code that creates warnings in the JVM, but none of them caused any real errors in our testing.
- **Grammar Generation:** ANTLR 2.7.6
  ANTLR was used for both the Lexer and the Parser. It was also used to create the AST necessary for the final code generation. The 2.7.6 branch of ANTLR is the default version that comes with the Eclipse ANTLR plugin.
- **Version Control:** CVS
  CVS was chosen as the version control system for the project due to its native integration with Eclipse. We relied on CVS for version history and group synchronization.

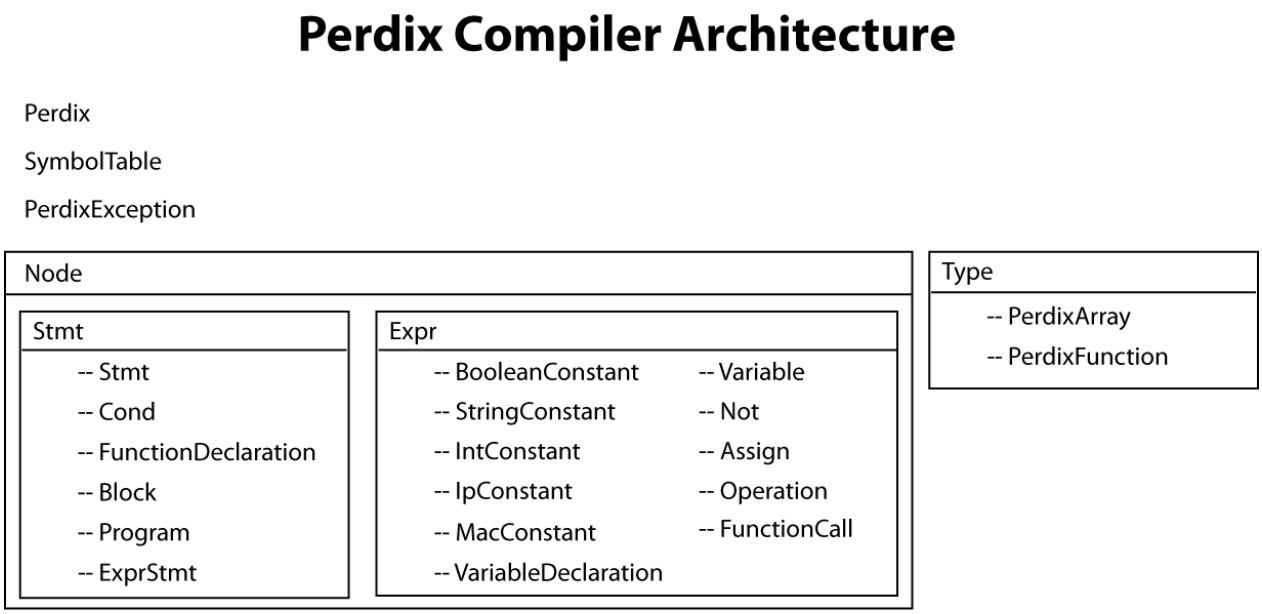
## 4.5 Project Timeline

The following shows the dates of the project milestones that were achieved. Project milestone dates were not firm and were subject to change. The language specifications went through many revisions before the final version was decided upon, which introduced unexpected delays into the project lifecycle. All dates listed are 2007.

| Date | Milestone |
|---|---|
| September 13 | Group Formed |
| September 25 | Project Proposal |
| October 18 | Language Reference Manual Completed |
| October 31 | Major revision to language - new features added |
| November * | Java Class Implementations and Code Generation |
| December 2 | Tree Walker completed |
| December 9-17 | Testing and Bug Fixes |
| December 18 | Final Release and Presentation |

# 5 Architectural Design

The following block diagram shows the Java classes involved in the structure of the Perdix compiler. The Node, Stmt, and Expr classes were created for hierarchy. The classes contained within the blocks extend or implement the class listed at the top of the box.




There are four main parts to the Perdix compiler. The first is the Perdix frontend, which allows the user to specify a command line argument for the input file to be compiled. Once the input file has been read, it is passed to the ANTLR grammar (PerdixGrammar.g) which contains both the lexer and the parser. The lexical analysis stage of the compiler tokenizes the input. The output of the lexer is passed to the parser, which consumes the tokens and checks for syntax errors. If an error is found, an error message is generated and the program terminates. However, if the input file passes the parsing stage, it is passed to the semantic analysis part of the compiler. This is the ANTLR tree walker, which consumes the abstract syntax tree (AST) generated by the parser. The tree walker contains a set of rules that call the individual Java classes listed in the diagram above. Each of these classes contains two main functions, a typeCheck(), which tests the input file for semantic meaning and a gen() function which translates the Perdix code into Java syntax. The typeCheck() is the last phase of the compiler where errors may be generated. If the input file typeChecks correctly, Java code is generated, which can be compiled and executed on a JVM.

Although many compilers contain a code optimization step, this was not within the scope of this project. A code optimizer could be added to streamline the Java output in future revisions.

# 6 Test Plan

## 6.1 Test Data

Testing was done using iptables firewall logs obtained from the Honeynet Project's Scan of the Month challenge. We used the data from Scan 30, which provided us with a full month of sanitized iptables logs from a server on the Internet (http://www.honeynet.org/scans/scan30/).

# 7 Lessons Learned

## 7.1 Justin Prosco

I have two written compilers in the past, but have never used ANTLR to create the grammar. I found ANTLR to be a bit confusing, but it did help by pointing out non-determinism in the grammars that we generated. This would not have been possible if we had developed our own parser and AST walker.

While I did learn how to integrate ANTLR into a project, I feel that the most important lessons that I learned were not technical, but rather, came from from working in a group. There were several obstacles that we encountered, from language barriers to scheduling conflicts and even submissions to CVS that broke the builds. As the group leader, I tried to keep everyone motivated and set reasonable deadlines for the project milestones to be achieved. This proved to be a difficult task however, as PLT is not the only class our team members had to focus on, and as a student, I do not have any authority to tell others what to do.

Despite several major setbacks in the project, I was able to stay motivated and do what what was necessary in order to achieve a successful outcome on the project. There will always be obstacles in any project, but the true value of this semester was the experience in dealing with those obstacles without losing sight of the final goal.

## 7.2 Angelika Zavou

I would never have expected that creating even a simple programming language and its compiler, could be such a difficult task. Apart from in depth knowledge of compilers concepts it also requires high-level programming skills and a lot of time.

Working on Lexer, Parser and Walker helped me understand in depth the concepts of a compiler (tokens, syntactic errors etc) and learn how to code using ANTLR, a completely different tool from what I had ever used in the past. Also, working intensely on the code generation and testing, significantly improved my coding skills in Java.

Another lesson I learned is that there may be many revisions to initial design concepts in such large. Even though we had settled the overall design early on, many times we needed to revise some of the decisions we made in other to facilitate the implementation. An example of this is the multiple versions of the walker and the grammar that we created. All this made me understand which of my early design decisions were not optimal and this will surely help me in my future projects.

The last lesson I took away from this project was the experience of working with a team on a big project. Regular group meetings, communication among team members, as well as team cooperation are all necessary for the successful completion of the project.

### 7.3 Orr Bibring

When I first began this project, I possessed virtually no knowledge about compilers. I knew I needed them to make my programs run, but the gritty details were embedded in a mysterious lock-box I never thought Id open. For me, this course has not only de-mystified this once abstract concept, but has also taught me a great deal more about the interplay between programming languages and computer capabilities.

More specifically, this course has taught me numerous technical and worldly lessons. From the technical standpoint, I learned about the construction of compilers: how they recognize the words I write (lexical analysis), how they build connections between the words I write (parsing), and how they evaluate the words I write (walking). I also learned about their specifications: the look ahead and its grammar restrictions, context free grammars as their languages, and non-determinism; among many other things. I also learned about the specifics of ANTLR: how to use it to create a lexer, parser, and treewalker to simplify much otherwise required work. I learned lots more about Java, including what I now dub my favorite operator: instanceof, and about eclipse for group programming projects.

From a worldly perspective, one of the most fascinating things I felt I took away from the course was the notion that programming languages are not just to be used, but are to be built. Prior to this course, this had never occurred to me. I had always assumed that building a programming language (and something to compile it) was a task far too advanced for an intermediate programmer. This revelation has really developed the meaning of the word powerful in my mind as it relates to programming.

I also learned a great deal about coding in a group setting. I now understand that commenting your code is truly critical, although I would once have scoffed at the notion. I also learned much more than I ever would have imaged about communication with others. Its certainly been an educational experience.

### 7.4 Bing Wu

I took the course Principles of Compiler and Translator when undergraduate, and did a C-like Complier, which involving lexical analysis, syntax analysis, semantic tree, three-address codes generation and Interpreter. This Translator project is different from my former compiler project, instead of intermediate code generation and interpreter, we translate our language directly to java language. Therefore I learn lots of new concepts about translator in this project. And get to know how to use tools to do the lexical, syntax and semantic analysis, and building tree, I used to do this by writing code myself without tools. Besides, I have lots of fun in the part of implementing of multi-dimensional array in our language using nesting hashtable, which is very interesting. I enjoy the process to think and figure out, which cost me a whole day to perfect it. I translate the simple multi-dimensional array element to complicated java code of nesting hashtable, and I wrote a ArrayIndicesHandling class to automatically tell where should create a new hashtable in any level. This ArrayIndicesHandling thing is like a more complicated symbol table.

## 8 Current Limitations

Perdix is currently limited to only parsing `iptables` log files and is not able to parse logs in other formats. This is because the keywords have been highly customized to the keywords found in `iptables` logs. The language could, however, be easily extended to include support for other text file log formats in the future.

# A Example Programs

This appendix provides example programs in the Perdix language. This section should be used as a reference for available functionality.

This simple example illustrates the use of variables and various print methods.

```
# Example1:  Variable declaration
#          - Integer arithmetics
#          - Assignments
#          - Print, Open, Close

function start()
{
    int port1 = 80;
    file output = "output.txt";
    ip4address ip1 = @211.168.230.94;
    mac mac1 = 00:00:00:ab:ab:ab;
    int port2 = 96;
    string str1 = "The results are :";

    array int a[int][string];

    int sum = 0;

    # multi-dimensional arrays
    a[0][str1] = 2;
    a[1]["test"] = 3;

    sum = a[0][str1] + 100;

    open(output, "w");

    print(output, "Port 2: ", port2);
    print(screen, "Port 2: ", port2);
    print(screen, a);
    print(screen, sum);
    port2 = port1 + 1;
    print(screen, str1, port1, port2);
    close(output);
}
```

This program highlights the creation of user-defined functions.

```
# Sample Perdix program

function findDstVal(string src_file, string dst_file, ip4address dst_val)
{
    file log = src_file;
    file QueryResults = dst_file;

    open (log, "r");
```

```
    open(QueryResults, "w");
    cond(log: DST == dst_val)
    {
        results(QueryResults, MAC);
    }
    close(log);
    close(QueryResults);
}

function start()
{
   file log = "Samples.txt";
   array string results_array[string];
   ip4address dst_val = @11.11.11.80;
   string eth1 = "eth1";

   #open file for reading
   open(log, "r");

   #call user-defined function "findDstVal"
   findDstVal("Samples.txt", "Perdix_Results.txt", dst_val);

   #use cond() to query the log file
   cond(log: PHYSOUT=="eth1" && RES=="0")
   {
      count(results_array, PHYSOUT);
      print(screen, RES);
   }
}
```

The program below illustrates the use of the write and results function. It also uses associative arrays.

```
function start()
{
    int a = 80;
    file input = "Samples.txt";
    file output = "output.txt";
    file res = "results.txt";

    ip4address ip1 = @211.168.230.94;
    string str1 = "something";
    mac mac1 = 00:00:00:ab:ab:ab;
    array ip4address a1[ip4address];

    open(input, "r");
    open(output, "w");
    open(res, "w");

cond(input : DST==ip1 && DPT >= 80)
{
    # write the unique values of DST to output
    unique(output,DST);
```

```
    # fill a1 with the SRC values from results
    write(a1,SRC);

    # print results to results file
    results(res, DATE, DST, SRC);

    # print the value of a1 to the screen
    print(screen,a1);

    # print the value of ip1 to the screen
    print(screen,ip1);
}

    close(input);
    close(output);
    close(res);
}
```

# B Netfilter Log Format

The section provides an overview of the fields available in the netfilter log file format. This is the standard log format used by iptables.

| Token | Column Name | Description |
|---|---|---|
| 1 | `Syslog prefix` | Timestamp from Syslog (Date) |
| 2 | `User defined prefix` | User-defined string |
| 3 | `IN` | Interface the packet was received on |
| 4 | `PHYSIN` | Physical interface the packet was received on |
| 5 | `OUT` | Interface the packet was sent on |
| 6 | `PHYSOUT` | Physical interface the packet was sent on |
| 7 | `MAC` | Destination MAC address |
| 8 | `SRC` | Source IP address of packet |
| 9 | `DST` | Destination IP address of packet |
| 10 | `LEN` | Length of IP packet in bytes |
| 11 | `TOS` | Type of service - type field |
| 12 | `PREC` | Type of service - precedence field |
| 13 | `TTL` | Remaining time to live of packet |
| 14 | `ID` | Unique ID for IP datagram |
| 15 | `CE` | Congestion experienced flag |
| 16 | `DF` | Don't fragment flag |
| 17 | `PROTO` | Protocol name or number |
| 18 | `SPT` | Source port (TCP and UDP) |
| 19 | `DPT` | Destination port (TCP and UDP) |
| 20 | `SEQ` | Sequence number |
| 21 | `WINDOW` | TCP reservation window size |
| 22 | `RES` | Reserved bits |
| 23 | `SYN` | TCP SYN flag |
| 24 | `ACK` | Acknowledgment flag |
| 25 | `PSH` | Push flag |
| 26 | `RST` | Reset flag |
| 27 | `FIN` | FIN flag |
| 28 | `URGP` | Urgent Pointer for out of band data trasnfer |

```
header {package perdix.parser;}

/*-------------------------------------------*/
/*               PERDIX LEXER                */
/*-------------------------------------------*/

class PerdixLexer extends Lexer;
options
        {k = 2;
        testLiterals = false;
        exportVocab = Perdix;
        charVocabulary = '\3'..'\377';}

/* Punctuation */
LPAREN : '(';
RPAREN : ')';
LBRACE : '{';
RBRACE : '}';
LBRACKET : '[';
RBRACKET : ']';
DOT: '.';
COMMA : ',';
DQUOTE : '"';
SEMI : ';';
COLON : ':';
DOLLAR : '$';
AT: '@';

/* Operators */
NOT: '!';
AND: "&&";
OR: "||";

EQ : "==";
NEQ : "!=";
GT : '>';
GEQ : ">=";
LT : '<';
LEQ : "<=";

ASSIGN : '=';
PLUS :'+';
MINUS: '-';


protected LETTER: ('a'..'z' | 'A'..'Z');
protected DIGIT: '0'..'9';
protected HEX: ('0'..'9' | 'a'..'f' | 'A'..'F');

/* Identifiers */
/* Numbers, MAC or IP Address */
NUMS options {testLiterals = true;}: (HEX HEX COLON) => HEX HEX COLON HEX HEX
     COLON HEX HEX COLON HEX HEX COLON HEX HEX COLON HEX HEX {$setType(MAC_ADDRE
SS);}
     | (DIGIT)+ {$setType(NUMBER);}
     | (LETTER |'_') (LETTER | DIGIT | '_')* {$setType(ID);};

IP_ADDRESS: (AT '1' DIGIT DIGIT) => AT '1' DIGIT DIGIT OCTET OCTET OCTET
     | (AT '2' ('0'..'4') DIGIT) => AT '2' ('0'..'4') DIGIT OCTET OCTET OCTET
     | (AT "25" ('0'..'5')) => AT "25" ('0'..'5') OCTET OCTET OCTET
     | AT (DIGIT)? DIGIT OCTET OCTET OCTET;

OCTET: (DOT '1' DIGIT DIGIT) => DOT '1' DIGIT DIGIT |
    (DOT '2' ('0'..'4') DIGIT) => DOT '2' ('0'..'4') DIGIT
    | (DOT "25" ('0'..'5')) => DOT "25" ('0'..'5') | DOT (DIGIT)? DIGIT;

/* String */
LITERAL: '"' ('"'! '"' | ~('"' | '\n'))* '"' ;
```



```
/* Whitespace */
WS: (' ' | '\t')+ {$setType(Token.SKIP);};

NEWLINE : ('\n' | ('\r' '\n') => '\r' '\n' | '\r') {
$setType(Token.SKIP); newline();};

/* Single-line comment */
COMMENT: '#' (~('\n'|'\r'))*
        {$setType(Token.SKIP);};


/*-------------------------------------------*/
/*              PERDIX PARSER                */
/*-------------------------------------------*/

class PerdixParser extends Parser;

options {k = 3;
                buildAST = true;
                exportVocab = Perdix;
                }

tokens {
    /* Imaginary nodes*/
    PROGRAM;
    BODY;
    EXPR_STMT;
    VAR_DECL_LIST;
    VAR_DECL;
    ARRAY;
    INDICES_LIST;
    STMT_LIST;
    FUNC_PARAM_LIST;
    FUNC_DECL_LIST;
    FUNC_CALL;
    PARAM_LIST;
    CONDFUNC_LIST;
}


/* Structure of Perdix programs */
program: func_decl_list
    {#program = #([PROGRAM, "PROGRAM"], program);};

/* Body of start() and user-defined functions */
body: LBRACE! var_decl_list stmt_list RBRACE!
{#body =#([BODY, "BODY"],body);};

func_decl_list: (func_decl)*
        {#func_decl_list = #([FUNC_DECL_LIST,"FUNC_DECL_LIST"], func_decl_list);
};

var_decl_list: (var_decl)*
        {#var_decl_list = #([VAR_DECL_LIST,"VAR_DECL_LIST"], var_decl_list);};

stmt_list: (stmt)*
        {#stmt_list = #([STMT_LIST, "STMT_LIST"], stmt_list);};

/* Functions declaration */
func_decl: "function"^ ID LPAREN! func_par_list RPAREN! body;

/* Parameter list in the function definition - a function with an empty
 * parameter list is also valid */
func_par_list: (var_decl_bare (COMMA! var_decl_bare)*)?
    {#func_par_list = #([FUNC_PARAM_LIST, "FUNC_PARAM_LIST"], func_par_list);};

param: type ID
    | "array" type ID indices_list;
```

```
type: ("int" | "string" | "file" | "ip4address" | "mac");

/* The body includes assigment statements, function call statements,
 * file open() and close() statements, cond() statements and print() statements
*/
stmt: expr_stmt
    | condition_func;

expr_stmt: expr SEMI!
                {#expr_stmt = #([EXPR_STMT, "EXPR_STMT"], expr_stmt);};

/* Variable Declaration */
var_decl_bare : (type ID (ASSIGN! cond_expr)?
                        | array_decl )
                        {#var_decl_bare= #([VAR_DECL, "VAR_DECL"], #var_decl_bar
e);}
                        ;

array_decl : "array"! type ID indices_list
       ;

var_decl : var_decl_bare SEMI!;

indices_list: (LBRACKET! type RBRACKET!)+
                ;


/* Associative Arrays */
assoc_array_decl: "assoc"^ ID LBRACKET! RBRACKET! SEMI!;

assoc_array_element: DOLLAR ID LBRACKET! (ID|NUMBER|LITERAL|IP_ADDRESS|MAC_ADDRE
SS) RBRACKET!;



/* Function Call Statement */
func_call: ID LPAREN! param_list RPAREN!
    { #func_call = #([FUNC_CALL, "FUNC_CALL"], func_call);};

param_list: (expr (COMMA! expr)*)?
    { #param_list = #([PARAM_LIST, "PARAM_LIST"], param_list);};


/* Conditional Statement */
condition_func: "cond"^ LPAREN! ID COLON! (cond_expr)? RPAREN! LBRACE! cond_body
 RBRACE!;

cond_body: (func_call SEMI!)*
    { #cond_body = #([CONDFUNC_LIST, "CONDFUNC_LIST"], cond_body);};



/* ---- EXPRESSIONS ---- */
expr : cond_expr (ASSIGN^ expr)?;

/* Logical Expressions*/
cond_expr: relat_expr ((AND^|OR^|NOT^) relat_expr)*;

/* Conditional Expressions*/
relat_expr : arithm_expr ((GT^ | GEQ^ | LT^ | LEQ^ | EQ^ | NEQ^) arithm_expr)*;


/* Arithmetic Expressions */
arithm_expr: arith_factor ((PLUS^ | MINUS^) arith_factor)*;

arith_factor :
    NOT^ factor | factor;

factor: ID^ (LBRACKET! expr RBRACKET!)*
```



```
    | LITERAL
    | MAC_ADDRESS
    | IP_ADDRESS
    | NUMBER
    | (LPAREN! cond_expr RPAREN!)
    | func_call;

/* ---- OPERATORS ---- */
/* Logical operators */
cond_op: NOT^ | AND^ | OR^;

/* Operators that apply only to integers */
int_op: GT^ | GEQ^ | LT^ | LEQ^ ;

/* Operators that apply to all data types */
general_op: EQ^ | NEQ^ ;

/* All operators except logical */
oper:  int_op
    | general_op;
```

```
header {package perdix.parser;}

{
        import java.io.*;
        import java.util.*;
        import perdix.core.*;
        import antlr.collections.AST;
}

class PerdixWalker extends TreeParser;

options
{
        importVocab = Perdix;
}

{
        // define empty symbol table for start of program
        SymbolTable top = new SymbolTable(null);
}

program returns [Program prog] throws PerdixException
{
        prog =null;
}
        : #(PROGRAM {prog = new Program();}
                        function_decl_list[prog]
                );

function_decl_list [Program prog] throws PerdixException
{
        FunctionDeclaration f1;
}
: #(FUNC_DECL_LIST (f1 = func_decl { prog.addFunctionDecl (f1); })*);

func_decl returns [FunctionDeclaration f] throws PerdixException
{
        Vector stmts_list;
        f = null;
        Stmt s = null;
        Vector v =null;
        Block block = new Block();
}
        : #("function"
                ID
                v = func_param_list
                block = func_body
                { f = new FunctionDeclaration(#ID.getText(), v, block); }
                );


func_body returns [Block block] throws PerdixException
{
        block = new Block();     // the block signifies a body definition
}
: #(BODY
        (variable_decls [block]{}
        statements_list [block]{})
        );

variable_decls [Block block] throws PerdixException
{
        VariableDeclaration var;
}
: #(VAR_DECL_LIST (var=variable_decl
        {block.addVariableDecl(var);})*)
        ;

variable_decl returns [VariableDeclaration v] throws PerdixException
```



```
{
        Type t = null;
        Type t1 = null;
        Vector indexTypes = new Vector();
        v = null;
        Expr a=null;
        Expr index = null;
        Vector indices;
}
: #(VAR_DECL t=type ID (t1 = type { indexTypes.insertElementAt(t1, 0); } )* (a=e
xpr)?
                {
                        for (ListIterator iter = indexTypes.listIterator();iter.
hasNext();)
                        {
                                t = new PerdixArray((Type)iter.next(), t);
                        }
                        v = new VariableDeclaration(#ID.getText(), t, a);
                }
        );


statements_list [Block block] throws PerdixException
{
        Stmt s1;
}
: #(STMT_LIST (s1=stmt
        {block.addStatement(s1);})*)
        ;

stmt returns [Stmt s] throws PerdixException
{
        s= null;
        Expr e1 = null, e2 = null;
        Expr a = null;
        String mode = null;
        String keyword = null;
        Vector v = null;
}
: #(EXPR_STMT (e1=expr {s= new ExprStmt(e1);})*)
 | s=condition_func;


condition_func returns [Stmt s] throws PerdixException
{
        s=null;
        Expr a = null;
        Vector v = new Vector();
}
        : #("cond" ID {Variable file=new Variable(#ID.getText());}
                  (a=expr)?
                   v = cond_body
                   {s=new Cond(file, a, v);});


cond_body returns [Vector v] throws PerdixException
{
        v = new Vector();     // the block signifies a body definition
        Expr fc = null;
}
: #(CONDFUNC_LIST
        (fc= func_call { v.add(fc); })*);


expr returns [Expr e] throws PerdixException
{
        Expr a, b;
        String str = null;
        Vector v = new Vector();    //for function arguments
```

```
        e = null;
}
        : #(PLUS a=expr b=expr) { e = new Operation("+", a, b); e.setLine(#PLUS.
getLine());e.setColumn(#PLUS.getColumn());}
        | #(MINUS a=expr b=expr) { e = new Operation("-", a, b); e.setLine(#MINU
S.getLine());e.setColumn(#MINUS.getColumn());}
        | #(EQ a=expr b=expr) { e = new Operation("==", a, b); e.setLine(#EQ.get
Line());e.setColumn(#EQ.getColumn());}
        | #(GT a=expr b=expr) { e= new Operation(">", a, b); e.setLine(#GT.getLi
ne());e.setColumn(#GT.getColumn());}
        | #(GEQ a=expr b=expr) { e= new Operation(">=", a, b); e.setLine(#GEQ.ge
tLine());e.setColumn(#GEQ.getColumn());}
        | #(LT a=expr b=expr) { e= new Operation("<", a, b); e.setLine(#LT.getLi
ne());e.setColumn(#LT.getColumn());}
        | #(LEQ a=expr b=expr) { e= new Operation("<=", a, b); e.setLine(#LEQ.ge
tLine());e.setColumn(#LEQ.getColumn());}
        | #(NEQ a=expr b=expr) { e = new Operation("!=", a, b); e.setLine(#NEQ.g
etLine());e.setColumn(#NEQ.getColumn());}
        | #(NOT a=expr) {e= new Not(a); e.setLine(#NOT.getLine()); e.setColumn(#
NOT.getColumn());}
        | #(OR a=expr b=expr) { e= new Operation("||", a, b); e.setLine(#OR.getL
ine());e.setColumn(#OR.getColumn()); }
        | #(AND a=expr b=expr) { e= new Operation("&&", a, b); e.setLine(#AND.ge
tLine());e.setColumn(#AND.getColumn());}
        | #(ASSIGN a=expr b=expr) {e= new Assign(a, b); e.setLine(#ASSIGN.getLin
e());e.setColumn(#ASSIGN.getColumn());}
        | NUMBER {e = new IntConstant(Integer.parseInt(#NUMBER.getText())); e.se
tLine(#NUMBER.getLine());e.setColumn(#NUMBER.getColumn());}
        | LITERAL {e = new StringConstant(#LITERAL.getText()); e.setLine(#LITERA
L.getLine());e.setColumn(#LITERAL.getColumn());}
        | MAC_ADDRESS {e = new MacConstant(#MAC_ADDRESS.getText()); e.setLine(#M
AC_ADDRESS.getLine());e.setColumn(#MAC_ADDRESS.getColumn());}
        | IP_ADDRESS {e =new IpConstant(#IP_ADDRESS.getText());e.setLine(#IP_ADD
RESS.getLine());e.setColumn(#IP_ADDRESS.getColumn());}
        | e=func_call
        | #(id:ID ( a=expr { v.add (a); })*
                {
                        e = new Variable(id.getText());
                        ((Variable)e).setIndices(v);

                }
        );


func_call returns [Expr e] throws PerdixException
{
        e = null;
}
        :#(FUNC_CALL {e = new FunctionCall(); Vector args = null;}
                ID {((FunctionCall)e).setFunctionName(#ID.getText());e.setLine(#
ID.getLine());e.setColumn(#ID.getColumn());}
                args=param_list {((FunctionCall)e).setArgs(args); }
                )
                ;

func_param_list returns [Vector v] throws PerdixException
{
        v = new Vector();
        VariableDeclaration t;
        }
        : #(FUNC_PARAM_LIST
                (t=variable_decl { v.add (t); })* );

type returns [Type t]
{
        t=null;
}
:       #("int"{ t=Type.PerdixInt;})
        | #("string" {t=Type.PerdixString;})
```



```
        | #("file" {t=Type.PerdixFile;})
        | #("mac" {t=Type.PerdixMac;})
        | #("ip4address" {t=Type.PerdixIp4address;})
        ;

param_list returns [Vector v] throws PerdixException
{
        v = new Vector();
        Expr e;
}
: #(PARAM_LIST(e=expr { v.add(e); })*);
```

```
package perdix.core;

public abstract class AbstractExpr
implements Expr{
        private int lineNumber;
        private int colNumber;

        public void setLine(int lineNumber)
        {
                this.lineNumber = lineNumber;
        }

        public void setColumn (int colNumber)
        {
                this.colNumber = colNumber;
        }

        public int getLine()
        {
                return this.lineNumber;
        }

        public int getColumn()
        {
                return this.colNumber;
        }
}
```

```
package perdix.core;

import java.util.ListIterator;

public class Assign extends AbstractExpr {

        private Type type;
        private Expr lhs, rhs;
        private Type ltype, rtype;

        public Assign(Expr lhs, Expr rhs)
        {
                this.lhs = lhs;
                this.rhs = rhs;
        }

        private String arrayTypeGen(Type t)
    {
            // if it's an int, return Integer
            // notice here we use "Integer", not "int", because this is inside J
ava generic
            if (t.equals(Type.PerdixInt)) return "Integer";
            // anything else other than an array will have type String
            if (!(t instanceof PerdixArray)) return "String";
            // now we are sure that it's an array, downcast it
            PerdixArray a = (PerdixArray)t;
            return "Hashtable< "
                            + arrayTypeGen(a.getKeyType())
                            + ","
                            + arrayTypeGen(a.getValueType()) +
                            " > ";
    }

        public Type typeCheck(SymbolTable st) throws PerdixException{

                ltype = lhs.typeCheck(st);
                rtype = rhs.typeCheck(st);


                if(!(ltype.equals(rtype)))
                        throw new PerdixException("Expression types do not match", this)
;



                // make sure that lhs of assignment is a variable
                if(!(lhs instanceof Variable))
                {
                        throw new PerdixException ("Cannot assign to a non-variable", thi
s);
                }

                this.type = ltype;

                ltype = st.get(((Variable)lhs).getName());

                return this.type;
        }

        public String gen() {

                /* typeCheck has already checked that lhs is a Variable */
                Variable var = (Variable)lhs;
                String output="";
                String lhs_output="";
                String lhs_output1 ="";
                String lhs_output2 ="";
                String lhs_output3 ="";
                String index_string ="";
```



```
                /* If it is not an array variable */
                if(var.indices.isEmpty())
                {
                        output = var.name + " = " + this.rhs.gen();
                }
                /* it is an array variable, so there is no assignment operator *
/
                else
                {
                        Type t = ltype;

                        for (ListIterator iter = var.indices.listIterator();iter
.hasNext();)
                        {
                                Expr index = (Expr)iter.next();

                                String full_index_string = "";
                                index_string = index.gen();

                                if (iter.hasNext())
                                {
                                        t = ((PerdixArray)t).getValueType();
                                        // generating guard check
                                        lhs_output1 = "if(" + var.name + lhs_outp
ut2 + ".get("+index_string+") == null)"+"\n";
                                        lhs_output3 = var.name + lhs_output2  +
".put("+index_string+", new "+ arrayTypeGen(t) +"());"+"\n";
                                        lhs_output2 += ".get("+index_string+")";
                                        output += lhs_output1 + lhs_output3;
                                }
                                else
                                {
                                        // the actual assignment
                                        output += var.name+lhs_output2 + ".put("+
index_string+","+rhs.gen()+ ")";
                                }
                        }
                }


                return output;
        }
}
```

```
package perdix.core;

import java.util.*;

public class Block implements Stmt{

        /* Vector for storing the statements in this block */
        private Vector stmts = new Vector();
        private Vector vardecs = new Vector();

        /* Constructor of an empty block */
        public Block ()
        {}
        public void addVariableDecl(VariableDeclaration vardec)
        {
                vardecs.add(vardec);
        }

        /* Adds a statement to the end of this block
        * @param stmt the new statement*/
        public void addStatement (Stmt stmt)
        {
                stmts.add(stmt);
        }

        public int size(){
                return stmts.size();
        }

        public Stmt getStatement(int index){
                return (Stmt)stmts.elementAt(index);
        }

        public Type typeCheck(SymbolTable st) throws PerdixException
        {
                ListIterator it = vardecs.listIterator();
                while (it.hasNext()) {
                        VariableDeclaration vardec  = (VariableDeclaration)it.ne
xt();
                        vardec.typeCheck(st);
                }
                ListIterator iter = stmts.listIterator();
                while (iter.hasNext()) {
                        Stmt stmt=(Stmt)iter.next();
                        stmt.typeCheck(st);
                }
                return Type.PerdixVoid;
        }

        public String gen() {
                ListIterator var_iter = vardecs.listIterator();
                ListIterator stmt_iter = stmts.listIterator();

                String output = "";

                /* Loop through the variable declarations and generate their cod
e */
                while (var_iter.hasNext())
                {
                        VariableDeclaration vardec = (VariableDeclaration)var_it
er.next();
                        output += vardec.gen();
                }

                /* Loop through the functions and generate their code */
                while(stmt_iter.hasNext())
                {
                        Stmt st = (Stmt)stmt_iter.next();
                        output += st.gen();
```



```
                }

                return output;
        }
}
```

```
package perdix.core;

public class BooleanConstant  extends AbstractExpr{

        private boolean value;

        public BooleanConstant()
        {

        }
        public BooleanConstant(boolean value)
        {
                this.value = value;
        }

        public boolean getValue()
        {
                return this.value;
        }

        public void setValue(boolean value)
        {
                this.value = value;
        }

        public String toString()
        {
                return Boolean.toString(this.value) ;
        }

        public Type getType()
        {
                return Type.PerdixBoolean;
        }

        public Type typeCheck(SymbolTable st)
        {
                return this.getType();
        }

        public String gen() {
                return this.toString();
        }
}
```

```
package perdix.core;

import antlr.*;
import antlr.collections.*;


public class CommonASTWithLine extends CommonAST{

                private int col = -1;
                private int line = -1;

                public void initialize(Token tok)
                {
                        super.initialize(tok);
                        line = tok.getLine();
                        col = tok.getColumn();
                }
                public void initialize (AST ast)
                {
                        super.initialize(ast);
                        if (ast instanceof CommonASTWithLine)
                        {
                                col = ((CommonASTWithLine)ast).getColumn();
                                line = ((CommonASTWithLine)ast).getLine();
                        }
                }
                public int getLine()
                {
                        return line;
                }
                public int getColumn()
                {
                        return col;
                }
}
```

```
package perdix.core;

import java.util.*;

public class Cond implements Stmt {

        private Variable filename;
        private Expr query;
        private Vector statements;

        public Cond(Variable file_id, Expr expr, Vector v)
        {
                this.filename = file_id;
                this.query = expr;
                this.statements = v;
        }

        public Type typeCheck(SymbolTable old) throws PerdixException
        {
                // create a new scope in this function
                SymbolTable st = new SymbolTable(old);

                // add the iptables keywords and their types to the symbol table
                st.put("DATE", Type.PerdixString);
                st.put("IN", Type.PerdixString);
                st.put("PHYSIN", Type.PerdixString);
                st.put("OUT", Type.PerdixString);
                st.put("PHYSOUT", Type.PerdixString);
                st.put("MAC", Type.PerdixMac);
                st.put("SRC", Type.PerdixIp4address);
                st.put("DST", Type.PerdixIp4address);
                st.put("LEN", Type.PerdixInt);
                st.put("TOS", Type.PerdixString);
                st.put("PREC", Type.PerdixString);
                st.put("TTL", Type.PerdixInt);
                st.put("ID", Type.PerdixInt);
                st.put("CE", Type.PerdixInt);
                st.put("DF", Type.PerdixInt);
                st.put("PROTO", Type.PerdixString);
                st.put("SPT", Type.PerdixInt);
                st.put("DPT", Type.PerdixInt);
                st.put("SEQ", Type.PerdixInt);
                st.put("WINDOW", Type.PerdixInt);
                st.put("RES", Type.PerdixString);
                st.put("SYN", Type.PerdixInt);
                st.put("ACK", Type.PerdixInt);
                st.put("PSH", Type.PerdixInt);
                st.put("RST", Type.PerdixInt);
                st.put("FIN", Type.PerdixInt);
                st.put("URGP", Type.PerdixInt);

                // add the built in functions to the symbol table
                st.put("results", new PerdixFunction(new Type[] {Type.PerdixFile,
Type.PerdixVoid}));
                st.put("unique", new PerdixFunction((new Type[] {Type.PerdixFile,
 Type.PerdixVoid})));
                st.put("write", new PerdixFunction(new Type[] {new PerdixArray(Ty
pe.PerdixString, Type.PerdixInt)}));
                st.put("count", new PerdixFunction(new Type[] {new PerdixArray(Ty
pe.PerdixString, Type.PerdixInt), Type.PerdixVoid}));

                // typecheck the file_id
                if(!(filename.typeCheck(st).equals(Type.PerdixFile)))
                        throw new PerdixException("first argument to cond must be a file");

                // typecheck the expr
                if (query != null)
                        query.typeCheck(st);
```



```
                // typecheck functions in the cond body
                for(int i=0; i<statements.size(); i++)
                {
                        if(!(statements.get(i) instanceof FunctionCall)) {
                                throw new PerdixException("only functions can be called wi
thin a cond() body");
                        }

                        ((FunctionCall)statements.get(i)).typeCheck(st);
                }

                // the cond function does not return a value
                return Type.PerdixVoid;
        }

        public String gen()
        {
                String indent = "     ";

                // vectorize the file so that it can be passed to functions
                String output = "\n\n    while(" + this.filename.getName() + "_stream.
ready()) {\n";
                output += indent + "line=" + this.filename.getName() + "_stream.read
Line();\n";

                // parse the line and set variables to the value contained
                output += indent +"String[] tokens = line.trim().split(\"\\\\s+\");\n";

                output += indent +"DATE = \"\";\n";
                output += indent +"IN = \"\";\n";
                output += indent +"PHYSIN = \"\";\n";
                output += indent +"PHYSOUT = \"\";\n";
                output += indent +"MAC = \"\";\n";
                output += indent +"OUT = \"\";\n";
                output += indent +"SRC = \"\";\n";
                output += indent +"DST = \"\";\n";
                output += indent +"TOS = \"\";\n";
                output += indent +"PREC = \"\";\n";
                output += indent +"RES = \"\";\n";
                output += indent +"LEN = 0;\n";
                output += indent +"TTL = 0;\n";
                output += indent +"ID = 0;\n";
                output += indent +"CE = 0;\n";
                output += indent +"SPT = 0;\n";
                output += indent +"DPT = 0;\n";
                output += indent +"SEQ = 0;\n";
                output += indent +"WINDOW = 0;\n";
                output += indent +"SYN = 0;\n";
                output += indent +"ACK = 0;\n";
                output += indent +"PSH = 0;\n";
                output += indent +"RST = 0;\n";
                output += indent +"FIN = 0;\n";
                output += indent +"URGP = 0;\n\n";

                // parse the "date"
                output += indent +"DATE = tokens[0] + \" \" + tokens[1] + \" \" + tokens[2];\n";

                output += indent +"for(int i=3; i<tokens.length; i++) {\n";
                output += indent +"currentToken = tokens[i];\n";

                output += indent +
                          "if(currentToken.startsWith(\"IN=\") == true)\n";
                output += indent +
                          "IN = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.length
());\n";
                output += indent +
                          "else if(currentToken.startsWith(\"PHYSIN=\") == true)\n";
                output += indent +
```

```
                        "PHYSIN = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.
length());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"PHYSOUT=\") == true)\n";
                output += indent +
                        "PHYSOUT = currentToken.substring(currentToken.indexOf(\"=\")+1,currentTok
en.length());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"MAC=\") == true)\n";
                output += indent +
                        "MAC = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.len
gth());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"OUT=\") == true)\n";
                output += indent +
                        "OUT = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.leng
th());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"SRC=\") == true)\n";
                output += indent +
                        "SRC = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.leng
th());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"DST=\") == true)\n";
                output += indent +
                        "DST = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.leng
th());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"TOS=\") == true)\n";
                output += indent +
                        "TOS = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.leng
th());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"PREC\") == true)\n";
                output += indent +
                        "PREC = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.len
gth());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"RES\") == true)\n";
                output += indent +
                        "RES = currentToken.substring(currentToken.indexOf(\"=\")+1,currentToken.leng
th());\n";
                output += indent +
                        "else if(currentToken.startsWith(\"LEN=\") == true)\n";
                output += indent +
                        "LEN = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,c
urrentToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"TTL=\") == true)\n";
                output += indent +
                        "TTL = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,cu
rrentToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"ID=\") == true)\n";
                output += indent +
                        "ID = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,curr
entToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"SPT=\") == true)\n";
                output += indent +
                        "SPT = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,cu
rrentToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"DPT=\") == true)\n";
                output += indent +
                        "DPT = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,cu
rrentToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"WINDOW=\") == true)\n";
```



```
                output += indent +
                        "WINDOW = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\"
)+1,currentToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"SEQ=\") == true)\n";
                output += indent +
                        "SEQ = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,cu
rrentToken.length()));\n";
                output += indent +
                        "else if(currentToken.startsWith(\"URGP=\") == true)\n";
                output += indent +
                        "URGP = Integer.parseInt(currentToken.substring(currentToken.indexOf(\"=\")+1,
currentToken.length()));\n";
                output += indent +
                        "else if(currentToken.compareTo(\"CE\") == 0)\n";
                output += indent +
                        "CE = 1;\n";
            output += indent +
                "else if(currentToken.compareTo(\"SYN\") == 0)\n";
                output += indent +
                        "SYN = 1;\n";
                output += indent +
                        "else if(currentToken.compareTo(\"ACK\") == 0)\n";
                output += indent +
                        "ACK = 1;\n";
                output += indent +
                        "else if(currentToken.compareTo(\"PSH\") == 0)\n";
                output += indent +
                        "PSH = 1;\n";
                output += indent +
                        "else if(currentToken.compareTo(\"RST\") == 0)\n";
                output += indent +
                        "RST = 1;\n";
                output += indent +
                        "else if(currentToken.compareTo(\"FIN\") == 0)\n";
                output += indent +
                        "FIN = 1;\n";
                output += "    }\n";

                if(query != null)
                {
                        // add the item to the results vector with the query sta
tement
                        output += indent +"if(" + convertExpr(query.gen()) + ")\n"
;
                        output += indent +"results.add(line);\n";
                }
                else
                        // otherwise, add the whole line to the results vector
                        output += indent + "results.add(line);\n";

                output += "  }\n";

                // generate the statements in the cond body
                for(int i=0; i<statements.size(); i++)
                {
                        output += ((FunctionCall)statements.get(i)).gen();
                }

                return output;
        }

        /* converts the logical expression into a Java compatible query string *
/
        private String convertExpr(String expr) {

                // strip whitespace from the string
                expr = expr.replaceAll(" +", "");
```

```
                // format the not operators for strings
                expr = expr.replaceAll("DATE\\!=", "!DATE.equals(");
                expr = expr.replaceAll("PHYSIN\\!=", "!PHYSIN.equals(");
                expr = expr.replaceAll("PHYSOUT\\!=", "!PHYSOUT.equals(");
                expr = expr.replaceAll("MAC\\!=", "!MAC.equals(");
                expr = expr.replaceAll("OUT\\!=", "!OUT.equals(");
                expr = expr.replaceAll("SRC\\!=", "!SRC.equals(");
                expr = expr.replaceAll("DST\\!=", "!DST.equals(");
                expr = expr.replaceAll("TOS\\!=", "!TOS.equals(");
                expr = expr.replaceAll("PREC\\!=", "!PREC.equals(");
                expr = expr.replaceAll("RES\\!=", "!RES.equals(");

                // format the equals operators for strings
                expr = expr.replaceAll("DATE==", "!DATE.equals(");
                expr = expr.replaceAll("PHYSIN==", "!PHYSIN.equals(");
                expr = expr.replaceAll("PHYSOUT==", "!PHYSOUT.equals(");
                expr = expr.replaceAll("MAC==", "!MAC.equals(");
                expr = expr.replaceAll("OUT==", "!OUT.equals(");
                expr = expr.replaceAll("SRC==", "!SRC.equals(");
                expr = expr.replaceAll("DST==", "!DST.equals(");
                expr = expr.replaceAll("TOS==", "!TOS.equals(");
                expr = expr.replaceAll("PREC==", "!PREC.equals(");
                expr = expr.replaceAll("RES==", "!RES.equals(");


                // format the operators for Java output
                //expr = expr.replaceAll("==", ".startsWith(");
                expr = expr.replaceAll("&&", ")&&");
                expr = expr.replaceAll("\\|\\|", ")||");
                expr = expr.replaceAll("<", "<(");
                expr = expr.replaceAll(">", ">(");
                expr = expr.replaceAll("<\\(=", "<=(");
                expr = expr.replaceAll(">\\(=", ">=(");

                // complete the expression
                expr += ")";

                return expr;
        }
}
```

```
package perdix.core;

public interface Expr extends Node {
        public void setLine(int lineNumber);
        public void setColumn (int colNumber);
        public int getLine();
        public int getColumn();
}
```

```
package perdix.core;

public class ExprStmt implements Stmt{

        private Expr expr = null;

        public ExprStmt()
        {
        }

        public ExprStmt(Expr expr)
        {
                this.expr = expr;
        }

        public Type typeCheck(SymbolTable st) throws PerdixException
        {
                this.expr.typeCheck(st);

                return Type.PerdixVoid;
        }

        public String gen()
        {
                return "    " + this.expr.gen()+ ";";
        }

}
```

```
package perdix.core;

import java.util.*;

public class FunctionCall extends AbstractExpr {

        private String functionName = null;
        private Vector args = new Vector();

        public FunctionCall ()
        {}

        public FunctionCall (String functionName, Vector args)
        {
                this.functionName = functionName;
                this.args = args;
        }

        public void setFunctionName (String functionName)
        {
                this.functionName = functionName;
        }

        public String getFunctionName ()
        {
                return this.functionName;
        }

        public void setArgs (Vector args)
        {
                this.args = args;
        }

        public Vector getArgs ()
        {
                return this.args;
        }


        public Type typeCheck(SymbolTable st) throws PerdixException
        {
                Type t = st.get(this.functionName);
                Type argtype;

                // v stores the types obtained from the symbol table
                Vector v = ((PerdixFunction)t).args;

                if(!(t instanceof PerdixFunction))
                {
                        throw new PerdixException(this.functionName + " is not a defi
ned funciton", this);
                }

                // built-in functions are checked first for special cases

                // print has an unlimited number of arguments, so it must be typ
echecked before
                // checking the number of arguments
                if(this.functionName.compareTo("print") == 0)
                {
                        if(st.get(((Variable)args.get(0)).getName()) == null)
                                throw new PerdixException("argument not declared", thi
s);

                        if(!((Type)v.get(0)).equals(st.get(((Variable)args.get(0
)).getName())))
                                throw new PerdixException("print() first argument must be o
f type file", this);
```



```
                        if(args.size() < 2)
                                throw new PerdixException("incorrect number of arguments
passed to print()", this);

                        // cannot compare argument size, as variable_list has un
known size
                        // no need to typecheck the rest, since they can be any
type
                        // so simply return from typeCheck with success
                        return Type.PerdixVoid;
                }

                // check the results function
                // unlimited number of arguments - same as above
                if(this.functionName.compareTo("results") == 0)
                {
                        if(st.get(((Variable)args.get(0)).getName()) == null)
                                throw new PerdixException("argument not declared", thi
s);

                        if(!((Type)v.get(0)).equals(st.get(((Variable)args.get(0
)).getName())))
                                throw new PerdixException("results() first argument must be
of type file", this);

                        // rather than checking types for the iptables keywords,
 check the argument names
                        checkIPTablesArgs(st);

                        return Type.PerdixVoid;
                }

                // check the write function
                // unlimited number of arguments - same as above
                if(this.functionName.compareTo("write") == 0)
                {
                        if(st.get(((Variable)args.get(0)).getName()) == null)
                                throw new PerdixException("argument not declared", thi
s);

                        if(!((Type)v.get(0)).equals(st.get(((Variable)args.get(0
)).getName())))
                                throw new PerdixException("write() first argument must be
of type array", this);

                        // rather than checking types for the iptables keywords,
 check the argument names
                        checkIPTablesArgs(st);

                        return Type.PerdixVoid;
                }


                // check for the correct number of arguments supplied to the fun
ction
                if(v.size() != args.size())
                {
                        throw new PerdixException("incorrect number of arguments passed to fu
nction " + this.functionName, this);
                }

                // check the open function
                else if(this.functionName.compareTo("open") == 0)
                {
                        if(st.get(((Variable)args.get(0)).getName()) == null)
                                throw new PerdixException("argument not declared", thi
s);

                        if(!((Type)v.get(0)).equals(st.get(((Variable)args.get(0
```

```
)).getName())))
                                throw new PerdixException("first argument must be a file to
open()", this);

                        if(!(this.args.get(1) instanceof StringConstant))
                                throw new PerdixException("file mode must be a constant"
, this);

                        if((((StringConstant)this.args.get(1)).getValue()).compa
reTo("w") < 0 &&
                                        (((StringConstant)this.args.get(1)).getV
alue()).compareTo("r") < 0)
                                throw new PerdixException("invalid file mode passed to ope
n()", this);
                }

                // check the unique function
                else if(this.functionName.compareTo("unique") == 0)
                {
                        if(st.get(((Variable)args.get(0)).getName()) == null)
                                throw new PerdixException("argument not declared", thi
s);

                        if(!((Type)v.get(0)).equals(st.get(((Variable)args.get(0
)).getName())))
                                throw new PerdixException("unique() first argument must b
e of type file", this);

                        // rather than checking types for the iptables keywords,
 check the argument names
                        checkIPTablesArgs(st);
                }

                // check the count function
                else if(this.functionName.compareTo("count") == 0)
                {
                        if(st.get(((Variable)args.get(0)).getName()) == null)
                                throw new PerdixException("argument not declared", thi
s);

                        if(!((Type)v.get(0)).equals(st.get(((Variable)args.get(0
)).getName())))
                                throw new PerdixException("count() first argument must be
of type array", this);

                        // rather than checking types for the iptables keywords,
 check the argument names
                        checkIPTablesArgs(st);
                }

                // check the arguments passed to generic functions
                else {
                        for(int i=0; i<args.size(); i++)

                        {
                                if(!((Type)v.get(i)).equals(((Expr)args.get(i)).
typeCheck(st)))

                                {
                                        throw new PerdixException("argument " + i
+ " of wrong type to " + this.functionName, this);
                                }
                        }
                }

                // function calls do not have a return value
                return Type.PerdixVoid;
        }

        // code generation
```



```
        public String gen()
        {
                String output="";

                // private static void Unique (String file_id, String keyword, V
ector results)
                if (this.functionName.compareTo("unique")==0)
                {
                        String fileid = ((Variable)args.get(0)).getName();
                        String keyword = ((Variable)args.get(1)).getName();

                        output += "\n    Unique(" + fileid + "," + "\"" + keyword +
 "\", results);";
                }

                // private static void Print(String file_id, Vector varlist)
                else if (this.functionName.compareTo("print")==0)
                {
                        output += "\n    print_list.clear();";

                        for(int i=1; i<args.size(); i++)
                        {
                                if (((this.args.get(i)) instanceof StringConstan
t))
                                        output += "\n    print_list.add(\"" + ((StringC
onstant)args.get(i)).getValue() + "\");";
                                else
                                        output += "\n    print_list.add(" + ((Variable
)args.get(i)).getName() + ");";
                        }

                        if (((Variable)args.get(0)).getName().equals("screen"))
                        {
                                output += "\n    Print(\"screen\", print_list);";
                        }
                        else if ((this.args.get(0)) instanceof StringConstant)
                        {
                                output += "\n    Print(" + ((StringConstant)args.ge
t(0)).getValue() + ",print_list);";
                        }
                        else
                        {
                                output += "\n    Print(" + ((Variable)args.get(0)).
getName() + ",print_list);";
                        }

                }

                // private static Vector Write(Vector results,Vector keywords)
                else if (this.functionName.compareTo("write")==0)
                {
                        output += "\n    keylist.clear();";

                        for (int i=1; i<args.size(); i++)
                        {
                                output += "\n    keylist.add(\"" + ((Variable)args.get
(i)).getName() + "\");\n";
                        }

                        // call the write built-in function
                        output += "    " + ((Variable)args.get(0)).getName() + "
= Write(results, keylist);";

                        return output;
                }

                // private static void CloseReader(BufferedReader in)
                // private static void CloseWriter(BufferedWriter out)
                else if (this.functionName.compareTo("close")== 0)
```

```
                {
                        String filename = ((Variable)args.get(0)).getName();

                        output += "\n    " + filename + "_stream.close();";
                }

                // private static BufferedReader OpenReader(String filename)
                // private static BufferedWriter OpenWriter(String filename)
                else if (this.functionName.compareTo("open")==0)
                {
                        String filename = ((Variable)args.get(0)).getName();
                        String mode = ((StringConstant)args.get(1)).getValue();

                        if(mode.compareTo("r") == 0) {
                                output += "\n    BufferedReader " + filename + "_stream
= OpenReader(" + filename + ");";
                        }
                        else if(mode.compareTo("w") == 0) {
                                output += "\n    BufferedWriter " + filename + "_stream
= OpenWriter(" + filename + ");";
                        }
                }

                // private static Hashtable Count (String keyword, Vector result
s)
                else if (this.functionName.compareTo("count")==0)
                {
                        output += "    " + ((Variable)args.get(0)).getName() + ".
clear();\n";

                        output += "    "+ ((Variable)args.get(0)).getName() +
                              " = Count(\"" + ((Variable)args.get(1)).getName() + "\"
, results);\n";
                }

                //private static void Results(String filename, Vector keylist, V
ector results)
                else if (this.functionName.compareTo("results")==0)
                {
                        output += "    keylist.clear();\n";

                        for (int i=1; i<args.size(); i++)
                        {
                                output += "    keylist.add(\"" + ((Variable)args.get(
i)).getName() + "\");\n";
                        }

                        output += "    Results(" + ((Variable)args.get(0)).getName(
) + ",keylist,results);\n";
                }
                else
                {
                        output += this.functionName + "(";

                        for(int i=0; i<args.size(); i++) {
                                output += ((Expr)args.get(i)).gen() + ",";
                        }

                        output = output.substring(0, output.length()-1);

                        output += ")";
                }

                return output;
        }


        // check the arguments passed to a function to ensure they are iptables
keywords
        // this typechecks the iptables arguments passed as a list to a function
```



```
        private void checkIPTablesArgs(SymbolTable st) throws PerdixException {
                String argname;

                String[] iptables_keys = { "DATE","IN","PHYSIN","PHYSOUT","MAC
","OUT","SRC",
                                                "DST","TOS","PREC","RES","LE
N","TTL","ID","CE",
                                                "SPT","DPT","SEQ","WINDOW",
"SYN","ACK","PSH","RST",
                                                "FIN","URGP"};

                // create an arraylist for more efficient searching
                ArrayList iptables_list = new ArrayList();

                for(int i=0; i<iptables_keys.length; i++) {
                        iptables_list.add(iptables_keys[i]);
                }

                // check all the arguments against the iptables keywords
                for(int i=1; i<args.size(); i++) {
                   argname = ((Variable)args.get(i)).getName();

                   if(!iptables_list.contains(argname))
                                throw new PerdixException("invalid keyword parameter passed to "
 + this.functionName, this);
                }
        }
}
```

```
/* The function data type */

package perdix.core;

import antlr.collections.AST;
import java.util.*;

public class FunctionDeclaration implements Stmt{

        private String functionName = null;
        private Vector args = new Vector();
        private Block body;

        public FunctionDeclaration ()
        {}

        public FunctionDeclaration(String functionName, Vector args, Block body)
        {
                this.functionName = functionName;
                this.args = args;
                this.body = body;
        }

        public void setFunctionName (String functionName)
        {
                this.functionName = functionName;
        }
        public String getFunctionName ()
        {
                return this.functionName;
        }

        public void setArgs (Vector args)
        {
                this.args = args;
        }
        public Vector getArgs ()
        {
                return this.args;
        }
        public void setBody (Block body)
        {
                this.body = body;
        }
        public Block getBody ()
        {
                return this.body;
        }

        public Type typeCheck(SymbolTable old) throws PerdixException
        {
                //need to check that it is not declared previously in the Symbol
Table
                //if not add it in the SymbolTable
                SymbolTable st = new SymbolTable(old);

                ListIterator iter = args.listIterator();

                Vector v = new Vector();

                while(iter.hasNext()) {
                        VariableDeclaration var = (VariableDeclaration)iter.next
();
                        st.put(var.getName(), var.getDeclaredType());

                        v.add(var.getDeclaredType());
                }

                // add function definitions to the symbol table
```



```
                // do not add the start() function, since it cannot be called re
cursively
                if(functionName.compareTo("start") < 0) {
                        old.put(functionName, new PerdixFunction(v));
                }

                // typeCheck the function body
                body.typeCheck(st);

                // functions return type void since they cannot have a return va
lue
                return Type.PerdixVoid;
        }

        // code generation
        public String gen()
        {
                String output = "";

                if (functionName.compareTo("start") == 0) {
                        output += "public static void main(String[] args) throws Exception {\n";

                        // variables used by cond() function
                        output += "     Vector results = new Vector();\n";       // used by
cond()
                        output += "     Vector print_list = new Vector();\n\n"; // used by pri
nt()
                        output += "     Vector keylist = new Vector();\n";       // used by
results() and write()
                        output += "     String line;\n";
                        output += "     String currentToken;\n";


                        // create variables for iptables keywords
                        output += "     String DATE = \"\";\n";
                        output += "     String IN = \"\";\n";
                        output += "     String PHYSIN = \"\";\n";
                        output += "     String PHYSOUT = \"\";\n";
                        output += "     String MAC = \"\";\n";
                        output += "     String OUT = \"\";\n";
                        output += "     String SRC = \"\";\n";
                        output += "     String DST = \"\";\n";
                        output += "     String TOS = \"\";\n";
                        output += "     String PREC = \"\";\n";
                        output += "     String RES = \"\";\n";
                        output += "     int LEN = 0;\n";
                        output += "     int TTL = 0;\n";
                        output += "     int ID = 0;\n";
                        output += "     int CE = 0;\n";
                        output += "     int SPT = 0;\n";
                        output += "     int DPT = 0;\n";
                        output += "     int SEQ = 0;\n";
                        output += "     int WINDOW = 0;\n";
                        output += "     int SYN = 0;\n";
                        output += "     int ACK = 0;\n";
                        output += "     int PSH = 0;\n";
                        output += "     int RST = 0;\n";
                        output += "     int FIN = 0;\n";
                        output += "     int URGP = 0;\n\n";
                }
                else {
                        String arguments = "";

                        output += "  public static void " + functionName+"(";

                        ListIterator iter = args.listIterator();

                        while(iter.hasNext()) {
                                VariableDeclaration var = (VariableDeclaration)i
```

```
ter.next();
                                arguments += var.gen();
                                arguments += ", ";
                        }

                        // format the arguments list for pretty printing
                        arguments = arguments.trim();
                        arguments = arguments.substring(0, arguments.length()-1)
;


                        output += arguments;

                        output += ") throws IOException {" ;

                        // variables used by cond() function
                        output += "     Vector results = new Vector();\n";       // used by
cond()
                        output += "     Vector print_list = new Vector();\n\n"; // used by pri
nt()
                        output += "     Vector keylist = new Vector();\n";       // used by
results() and write()
                        output += "     String line;\n";
                        output += "     String currentToken;\n";


                        // create variables for iptables keywords
                        output += "     String DATE = \"\";\n";
                        output += "     String IN = \"\";\n";
                        output += "     String PHYSIN = \"\";\n";
                        output += "     String PHYSOUT = \"\";\n";
                        output += "     String MAC = \"\";\n";
                        output += "     String OUT = \"\";\n";
                        output += "     String SRC = \"\";\n";
                        output += "     String DST = \"\";\n";
                        output += "     String TOS = \"\";\n";
                        output += "     String PREC = \"\";\n";
                        output += "     String RES = \"\";\n";
                        output += "     int LEN = 0;\n";
                        output += "     int TTL = 0;\n";
                        output += "     int ID = 0;\n";
                        output += "     int CE = 0;\n";
                        output += "     int SPT = 0;\n";
                        output += "     int DPT = 0;\n";
                        output += "     int SEQ = 0;\n";
                        output += "     int WINDOW = 0;\n";
                        output += "     int SYN = 0;\n";
                        output += "     int ACK = 0;\n";
                        output += "     int PSH = 0;\n";
                        output += "     int RST = 0;\n";
                        output += "     int FIN = 0;\n";
                        output += "     int URGP = 0;\n\n";
                }

                output += body.gen();

                output += "\n  }\n\n";

                return output;
        }

}
```

```
package perdix.core;

public class IntConstant extends AbstractExpr {

        private int value;

        public IntConstant() { this.value = 0; }

        public IntConstant(int value)
        {
                this.value = value;
        }

        public int getValue()
        {
                return this.value;
        }

        public void setValue(int value)
        {
                this.value = value;
        }

        public String toString()
        {
                return Integer.toString(this.value) ;
        }

        public Type getType()
        {
                return Type.PerdixInt;
        }

        public Type typeCheck(SymbolTable st)
        {
                return this.getType();
        }

        public String gen() {
                return this.toString();
        }
}
```

```
package perdix.core;

public class IpConstant extends AbstractExpr {

        private String value="";

        public IpConstant()
        {

        }
        public IpConstant(String value)
        {
                this.value = value;
        }

        public String getValue()
        {
                return this.value;
        }

        public void setValue(String value)
        {
                this.value = value;
        }

        public String toString()
        {
                // strip off the @ before the address
                return (this.value).substring(1);
        }

        public Type getType()
        {
                return Type.PerdixIp4address;
        }

        public Type typeCheck(SymbolTable st)
        {
                return this.getType();
        }

        public String gen() {
                return "\"" + this.toString() + "\"";
        }
}
```

```
package perdix.core;

public class MacConstant extends AbstractExpr {

        private String value="";

        public MacConstant() {}

        public MacConstant(String value)
        {
                this.value = value;
        }

        public String getValue()
        {
                return this.value;
        }

        public void setValue(String value)
        {
                this.value = value;
        }

        public String toString()
        {
                return this.value ;
        }

        public Type getType()
        {
                return Type.PerdixMac;
        }

        public Type typeCheck(SymbolTable st)
        {
                return this.getType();
        }

        public String gen() {
                return "\"" + this.toString() + "\"";
        }
}
```

```
package perdix.core;

public interface Node {

        public Type typeCheck(SymbolTable st) throws PerdixException;
        public String gen();

}
```

```
package perdix.core;

public class Not extends AbstractExpr {

        private final Type type = Type.PerdixBoolean;
        private Expr rhs;

        public Not(Expr rhs)
        {
                this.rhs = rhs;
        }

        public Type typeCheck(SymbolTable st) throws PerdixException{

                if(!((this.type).equals(rhs.typeCheck(st))))
                        throw new PerdixException("Incompatible type in not expression", th
is);

                return this.type;
        }

        public String gen() {
                String output = "!" + rhs.gen();

                // this needs to be changed

                return output;
        }


}
```

```
package perdix.core;

public class Operation extends AbstractExpr {

        private Type type;
        private String op;
        private Expr lhs, rhs;


        public Operation(String op, Expr lhs, Expr rhs)
        {
                this.op = op;
                this.lhs = lhs;
                this.rhs = rhs;
        }

        public Type typeCheck(SymbolTable st) throws PerdixException{

                Type ltype = lhs.typeCheck(st);
                Type rtype = rhs.typeCheck(st);


                if(!(ltype.equals(rtype)))
                        throw new PerdixException("Expression types do not match", this)
;

                /* Arithmetic Expression */
                if (op.compareTo("+") == 0 || op.compareTo("-") == 0)
                {
                        if (!(ltype.equals(Type.PerdixInt))) {
                                throw new PerdixException ("Arithmetic lvalue must be an
integer", this);
                        }

                        if (!(rtype.equals(Type.PerdixInt)))
                        {
                                throw new PerdixException ("Arithmetic rvalue must be an
integer", this);
                        }

                        this.type=Type.PerdixInt;
                }

                /* Logical Expression */
                else if(op.compareTo("&&") == 0 || op.compareTo("||") == 0)
                {
                        if(!(ltype.equals(Type.PerdixBoolean)) || !(rtype.equals
(Type.PerdixBoolean)))
                                throw new PerdixException("Logical expression types do no
t match", this);

                        this.type = Type.PerdixBoolean;
                }


                /* Relational Expression */
                else if (op.compareTo(">") == 0 || op.compareTo(">=") == 0 ||
                                op.compareTo("<") == 0 || op.compareTo("<=") ==
 0)

                {
                        if (!(ltype.equals(Type.PerdixInt)) && !ltype.equals(Typ
e.PerdixIp4address) &&
                                        !(ltype.equals(Type.PerdixMac)))
                        {
                                throw new PerdixException ("Not compatible type", thi
s);
                        }
```



```
                        this.type = Type.PerdixBoolean;
                }

                else if (op.compareTo("==") == 0 || op.compareTo("!=") == 0)
                {
                        if (!(ltype.equals(Type.PerdixInt)) && !ltype.equals(Typ
e.PerdixIp4address) &&
                                        !(ltype.equals(Type.PerdixMac)) && !ltyp
e.equals(Type.PerdixString))
                        {
                                throw new PerdixException ("Not compatible type", thi
s);
                        }

                        this.type = Type.PerdixBoolean;
                }

                return this.type;
        }

        public String gen() {

                String output = "(" + lhs.gen();
                output += " " + op + " ";
                output += rhs.gen() + ")";

                return output;
        }
}
```

```
package perdix.core;

import java.io.*;
import antlr.CommonAST;
import antlr.DumpASTVisitor;

import perdix.parser.*;

public class Perdix {

        public static void main(String args[]) throws Exception
        {
                if(args.length < 1) {
                        System.err.print("No input file specified.");
                        System.exit(1);
                }

                try {
                        String inputfilename = args[0].substring(0,args[0].index
Of("."));

                        BufferedReader in = new BufferedReader(new FileReader(ar
gs[0]));   // this should take cmd line args

                        //invoke the lexer
                        PerdixLexer lexer = new PerdixLexer(in);

                        //invoke the parser
                        PerdixParser parser = new PerdixParser(lexer);

                        // need for line number generation
                        parser.setASTNodeClass("perdix.core.CommonASTWithLine");

                        parser.program();

                        CommonAST ast = (CommonAST)parser.getAST();

                        // print the AST - uncomment to debug
                        //DumpASTVisitor visitor = new DumpASTVisitor();
                        //visitor.visit(ast);

                        //walk the Parse Tree and convert it to our format
                        PerdixWalker walker = new PerdixWalker();
                        Program prog = walker.program(ast);

                        // set the class name based on command line arguments
                        // if the filename contains a dot, remove it
                        if(args[0].contains(".")) {
                                prog.setInputFile(inputfilename);
                        }
                        else {
                                prog.setInputFile(args[0]);
                        }

                        //create an empty SymbolTable
                        SymbolTable st = new SymbolTable();

                        BufferedWriter out = new BufferedWriter(new FileWriter(i
nputfilename + ".java"));

                        // typecheck the program to check for semantic errors
                        prog.typeCheck(st);

                        // generate the Java code and file
                        out.write(prog.gen());

                        in.close();
                        out.close();
                }
```



```
                catch(PerdixException e)
                {
                        System.err.println(e.getMessage()
                                        + ((e.getSource() != null)?
                                                (" at line "
                                                + e.getSource().getLine()
                                                + ":"
                                                + e.getSource().getColumn()):"")
                                );
                }
                catch(IOException e)
                {
                        System.err.println(e.getMessage());
                }
        }
}
```

```
package perdix.core;

public class PerdixArray extends Type {

        public Type keyType;
        public Type valueType;

        public PerdixArray(Type kt, Type vt){
                super("array");
                keyType = kt;
                valueType = vt;
        }

        public boolean equals(Object obj) {
                if(!(obj instanceof PerdixArray))
                        return false;

                if(!this.keyType.equals(((PerdixArray)obj).getKeyType())) return
 false;
                if(!this.valueType.equals(((PerdixArray)obj).getValueType())) re
turn false;

                return true;
        }

        public Type getValueType()
        {
                return valueType;
        }

        public Type getKeyType()
        {
                return keyType;
        }

}
```

```
package perdix.core;

public class PerdixException extends Exception{

        private Expr source = null;

        PerdixException (String message)
        {
                super("ERROR: " + message);
        }

        public PerdixException (String message, Expr source)
        {
                super(message);
                this.source = source;
        }

        public Expr getSource()
        {
                return source;
        }
}
```

```
package perdix.core;

import java.util.Vector;

public class PerdixFunction extends Type {

        public Vector args;

        public PerdixFunction(Vector v){
                super("Type");
                args = v;
        }

        public PerdixFunction(Type[] t)
        {
                super("Type");

                args = new Vector();

                for(int i=0; i<t.length; i++)
                {
                        args.add(t[i]);
                }
        }

        public Type getArgType(int index)
        {
                return (Type)args.get(index);
        }
}
```

```
package perdix.core;

import java.util.*;

public class Program implements Stmt {

        private ArrayList functionDecls;
        private String inputfile;
        private int tabscope = 1;


        public Program()
        {
                this.functionDecls = new ArrayList();
        }

                public void addFunctionDecl(FunctionDeclaration fdl){
                functionDecls.add(fdl);
        }

        //the size represents how many functions we have in our program
        public int size() {
                return functionDecls.size();
        }

        public void setInputFile(String infile)
        {
                this.inputfile = infile;
        }

        public Type typeCheck(SymbolTable st) throws PerdixException
        {
                // put built-in functions in the symbol table
                st.put("open", new PerdixFunction(new Type[] {Type.PerdixFile, T
ype.PerdixString}));
                st.put("close", new PerdixFunction(new Type[] {Type.PerdixFile}))
;
                st.put("print", new PerdixFunction(new Type[] {Type.PerdixFile}))
;

                // store file variable for standard out in the symbol table
                st.put("screen", Type.PerdixFile);


                //instead of the above for loop we will Try with the ListIterato
r
                ListIterator iter = functionDecls.listIterator();

                boolean startFunctionExisted = false;

                while (iter.hasNext()) {
                        FunctionDeclaration funcDecl  = (FunctionDeclaration)ite
r.next();
                        funcDecl.typeCheck(st);


                        if (funcDecl.getFunctionName().equals("start")) startFunct
ionExisted = true;
                }

                // ensure start() was the last function declared
                if (!startFunctionExisted) throw new PerdixException ("function start
() must be defined");

                return Type.PerdixVoid;
        }

        public String gen()
        {
                String output = "/* Generated by Perdix v1.0 */\n\n";
```



```
                output += "import java.io.*;\n";
                output += "import java.util.*;\n\n";

                output += "public class " + inputfile + " {\n\n";


                /***** Generate Print function *****/
                tabreset();
                output += tab() + "private static void Print(String file_id, Vector varlist) {\n";
                indent();
                output += tab() + "if(file_id.compareTo(\"screen\") ==0) {\n";
                indent();
                output += tab() + "for(int i=0; i<varlist.size(); i++) {\n";
                indent();
                output += tab() + "System.out.println(varlist.get(i).toString());\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "else {\n";
                indent();
                output += tab() + "try {\n";
                indent();
                output += tab() + "BufferedWriter out = new BufferedWriter(new FileWriter(file_id))
;\n\n";
                output += tab() + "for(int i=0; i<varlist.size(); i++) {\n";
                indent();
                output += tab() + "out.write(varlist.get(i).toString());\n";
                outdent();
                output += tab() + "}\n\n";
                output += tab() + "out.close();\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "catch (IOException e) {\n";
                indent();
                output += tab() + "System.err.println(e);\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n\n";


                /***** Generate Results function *****/
                tabreset();
                output += tab() + "private static void Results(String filename, Vector keylist, Vector r
esults) {\n";
                indent();
                output += tab() + "try {\n";
                indent();
                output += tab() + "BufferedWriter out = OpenWriter(filename);\n";
                output += tab() + "String result = \"\";\n";

                /* set up the header of the csv */
                output += tab() + "for(int i=0; i<keylist.size(); i++) {\n";
                indent();
                output += tab() + "result += (String)keylist.elementAt(i) + \",\";\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "out.write(result.substring(0,result.length()-1) + \"\\n\\n\");\n";

                /* for each result output the keywords */
                output += tab() + "for(int j=0; j<results.size(); j++) {\n";
                indent();
                output += tab() + "String curr = (String)results.elementAt(j);\n";
                output += tab() + "Boolean found = false;\n";
                output += tab() + "result = \"\";\n";
                output += tab() + "for(int i=0; i<keylist.size(); i++) {\n";
```

```
                indent();
                output += tab() + "StringTokenizer st = new StringTokenizer(curr);\n";
                output += tab() + "String token = \"\";\n";
                output += tab() + "String key = (String)keylist.elementAt(i);\n";
                output += tab() + "if(key.compareTo(\"DATE\") == 0) {\n";
                indent();
                output += tab() + "for (int t=0; t<3; t++) {\n";
                indent();
                output += tab() + "token = token + st.nextToken() + \" \";\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "token = token.trim();\n";
                output += tab() + "result += token + \",\";\n";
                output += tab() + "found = true;\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "while(!found && (st.hasMoreTokens()))\n";
                output += tab() + "{\n";
                indent();
                output += tab() + "token = st.nextToken();\n";
                output += tab() + "if(token.contains(key)) {\n";
                indent();
                output += tab() + "if(token.contains(\"=\")) {\n";
                indent();
                output += tab() + "token = token.substring(token.indexOf(\"=\")+1);\n";
                outdent();
            output += tab() + "}\n";
                output += tab() + "else {\n";
                indent();
                output += tab() + "token = \"true\";\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "result += token + \",\";\n";
                output += tab() + "found = true;\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "if (!found) {\n";
                indent();
                output += tab() + "result += \",\";\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "found = false;\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "out.write(result.substring(0,result.length()-1) + \"\\n\\n\");\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "CloseWriter(out);\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "catch (IOException e) {\n";
                indent();
                output += tab() + "System.err.println(e);\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n\n";



                /***** Generate Unique function *****/
                tabreset();
                output += tab() + "private static void Unique (String file_id, String keyword, Vector r
esults) {\n";
                indent();
                output += tab() + "try {\n";
                indent();
                output += tab() + "Hashtable ht = new Hashtable(100);\n\n";
```



```
                output += tab() + "for(int i=0; i<results.size(); i++) {\n";
                indent();
                output += tab() + "String res = (String)results.elementAt(i);\n\n";
                output += tab() + "String s = \"\";\n";
                output += tab() + "StringTokenizer tok = new StringTokenizer(res);\n\n";
                output += tab() + "if(keyword.compareTo(\"DATE\") == 0) {\n\n";
                indent();
                output += tab() + "for (int t=0; t<3; t++) {\n";
                indent();
                output += tab() + "s += tok.nextToken() + \" \";\n";
                outdent();
                output += tab() + "}\n\n";
                output += tab() + "s = s.trim();\n\n";
                output += tab() + "if (!ht.containsKey(s)) { ht.put(s,1); }\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "else if(res.contains(keyword)) {\n";
                indent();
                output += tab() + "while(tok.hasMoreTokens()) {\n";
                indent();
                output += tab() + "s = tok.nextToken();\n\n";
                output += tab() + "if(s.contains(keyword)) {\n";
                indent();
                output += tab() + "if(s.contains(\"=\")) {\n";
                indent();
                output += tab() + "s = s.substring(s.indexOf(\"=\")+1,s.length());\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "else { s = keyword; } \n\n";
                output += tab() + "if (!ht.containsKey(s)) { ht.put(s,1); }\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "\n";
                output += tab() + "BufferedWriter out = OpenWriter(file_id);\n";
                output += tab() + "out.write(ht.toString());\n";
                output += tab() + "CloseWriter(out);\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "catch (IOException e) {\n";
                indent();
                output += tab() + "System.err.println(e);\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n\n";


                /***** Generate Count function *****/
                tabreset();
                output += tab() + "private static Hashtable Count (String keyword, Vector results) {\n
";
                indent();
                output += tab() + "Hashtable ht = new Hashtable(100);\n\n";
                output += tab() + "for(int i=0; i<results.size(); i++) {\n";
                indent();
                output += tab() + "String res = (String)results.elementAt(i);\n\n";
                output += tab() + "String s = \"\";\n";
                output += tab() + "StringTokenizer tok = new StringTokenizer(res);\n\n";
                output += tab() + "if(keyword.compareTo(\"DATE\") == 0) {\n\n";
                indent();
                output += tab() + "for (int t=0; t<3; t++) {\n";
                indent();
                output += tab() + "s += tok.nextToken() + \" \";\n";
```

```
                outdent();
                output += tab() + "}\n\n";
                output += tab() + "s = s.trim();\n\n";
                output += tab() + "if (!ht.containsKey(s)) { ht.put(s,1); }\n";
                output += tab() + "else {\n";
                indent();
                output += tab() + "int old = (Integer)ht.get(s);\n";
                output += tab() + "ht.remove(s);\n";
                output += tab() + "ht.put(s, ++old);\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "else if(res.contains(keyword)) {\n";
                indent();
                output += tab() + "while(tok.hasMoreTokens()) {\n";
                indent();
                output += tab() + "s = tok.nextToken();\n\n";
                output += tab() + "if(s.contains(keyword)) {\n";
                indent();
                output += tab() + "if(s.contains(\"=\")) {\n";
                indent();
                output += tab() + "s = s.substring(s.indexOf(\"=\")+1,s.length());\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "else { s = keyword; } \n\n";
                output += tab() + "if (!ht.containsKey(s)) { ht.put(s,1); }\n";
                output += tab() + "else {\n";
                indent();
                output += tab() + "int old = (Integer)ht.get(s);\n";
                output += tab() + "ht.remove(s);\n";
                output += tab() + "ht.put(s, ++old);\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                output += tab() + "return ht;\n";
                outdent();
                output += tab() + "}\n\n";


                /***** Generate Write function *****/
                tabreset();
                output += tab() + "private static Hashtable Write(Vector results,Vector keywords) {\n
";
                indent();
                output += tab() + "Hashtable output = new Hashtable();\n";
                output += tab() + "for(int i=0; i<results.size(); i++) {\n";
                indent();
                output += tab() + "String output_string = \"\";\n\n";
                output += tab() + "for(int j=0; j<keywords.size(); j++) {\n\n";
                indent();
                output += tab() + "String current_result = (String)results.get(i);\n";
                output += tab() + "String current_keyword = (String)keywords.get(j);\n";
                output += tab() + "StringTokenizer st = new StringTokenizer(current_result);\n\n";
                output += tab() + "if(current_keyword.compareTo(\"DATE\") == 0) {\n\n";
                indent();
                output += tab() + "for(int k=0; k<3; k++) {\n";
                indent();
                output += tab() + "output_string += st.nextToken() + \" \";\n";
                outdent();
                output += tab() + "}\n";
                outdent();
```



```
                output += tab() + "}\n";
                output += tab() + "else if(current_result.contains(current_keyword)) {\n";
                indent();
                output += tab() + "boolean found = false;\n\n";
                output += tab() + "while(!found) {\n";
                indent();
                output += tab() + "String curr_token = st.nextToken();\n\n";
                output += tab() + "if(curr_token.startsWith(current_keyword)) {\n";
                indent();
                output += tab() + "output_string += curr_token + \" \";\n";
                output += tab() + "found = true;\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n";
                outdent();
                output += tab() + "}\n\n";
                output += tab() + "output_string.trim();\n";
                output += tab() + "output.put(output_string,1);\n";
                outdent();
                output += tab() + "}\n\n";
                output += tab() + "return output;\n";
                outdent();
                output += tab() + "}\n\n";


                /***** Generate Open function *****/
                tabreset();
                output += tab() + "private static BufferedReader OpenReader(String filename) throws
 IOException {\n";
                indent();
                output += tab() + "return new BufferedReader(new FileReader(filename));\n";
                outdent();
                output += tab() + "}\n\n";

                tabreset();
                output += tab() + "private static BufferedWriter OpenWriter(String filename) throws I
OException {\n";
                indent();
                output += tab() + "return new BufferedWriter(new FileWriter(filename));\n";
                outdent();
                output += tab() + "}\n\n";


                /***** Generate Close function *****/
                tabreset();
                output += tab() + "private static void CloseReader(BufferedReader in) throws IOExce
ption {\n";
                indent();
                output += tab() + "in.close();\n";
                outdent();
                output += tab() + "}\n\n";

                tabreset();
                output += tab() + "private static void CloseWriter(BufferedWriter out) throws IOExce
ption {\n";
                indent();
                output += tab() + "out.close();\n";
                outdent();
                output += tab() + "}\n\n";


                ListIterator iter = functionDecls.listIterator();

                while (iter.hasNext()) {
                        FunctionDeclaration funcDecl  = (FunctionDeclaration)ite
r.next();
```

```
                        output += funcDecl.gen();
                }

                output += "}";

                return output;
        }

        /*
         * Tab formatting functions used for gen() output
         */

        // create a new tab scope
        private void indent()
        {
                this.tabscope++;
        }

        // return from a tab scope
        private void outdent()
        {
                this.tabscope--;
        }

        // print the number of tabs in the current scope
        private String tab()
        {
                String tabs = "";

                for(int i=0; i<this.tabscope; i++)
                {
                        tabs += "  ";
                }

                return tabs;
        }

        // reset the tab scope to the default value
        private void tabreset()
        {
                tabscope = 1;
        }
}
```

```
package perdix.core;

public interface Stmt extends Node
{

}
```

```
package perdix.core;

public class StringConstant extends AbstractExpr {

        private String value="";

        public StringConstant() {}

        public StringConstant(String value)
        {
                this.value = value;
        }

        public String getValue()
        {
                // strip the quotes from the value before returning
                return this.value.substring(1, this.value.length() -1);
        }

        public void setValue(String value)
        {
                this.value = value;
        }

        public String toString()
        {
                return this.value;
        }

        public Type getType()
        {
                return Type.PerdixString;
        }

        public Type typeCheck(SymbolTable st)
        {
                return this.getType();
        }

        public String gen() {
                return this.toString();
        }
}
```

```
package perdix.core;

import java.util.Hashtable;

/* The symbol table includes the binding between
   the name of a variable and its type */

public class SymbolTable {
        private Hashtable table;
        protected SymbolTable outer; //SymbolTable for enclosing scope

        public SymbolTable()
        {
                this(null);
        }

        public SymbolTable(SymbolTable st) {
                table = new Hashtable();
                outer = st;
        }

        //created the binding between the name and the type
        public void put(String name, Type t) throws PerdixException{
                // check for redeclare
                if (get(name) != null) throw new PerdixException("redeclaration of vari
able: + " + name);
                table.put(name, t);
        }

        public Type get(String name) {
                //Search in this and outer scopes for an identifier
                for (SymbolTable tab = this ; tab != null ; tab = tab.outer) {
                        Type found = (Type)(tab.table.get(name));
                        if (found != null) return found;
                        //the found==null is handled in class VariableDeclaratio
n
                }
                return null;
        }
}
```

```
package perdix.core;

public class Type {

        public final String name;

        protected Type (String s) {
                this.name = s;
        }

        protected boolean equals (Type t2)
        {
                if ((this.name).compareTo(t2.name) == 0) return true;
                else return false;
        }

        public static final Type
                PerdixInt = new Type("int"), PerdixString = new Type("string"),
                PerdixFile = new Type("file"), PerdixIp4address = new Type("ip4addr
ess"),
                PerdixMac = new Type("mac"), PerdixBoolean = new Type("boolean"),
                PerdixVoid = new Type("void");
}
```

```
/* The class for unsigned variables */

package perdix.core;

import java.util.*;

public class Variable extends AbstractExpr {

        public final String name;
        public Vector indices;

        public Variable(String name){
                this.name = name;
        }

        public void setIndices(Vector indices)
        {
                this.indices = indices;
        }

        public String getName() {
                return name;
        }

        //public int getIndexOfElement(String)

        public Type typeCheck(SymbolTable st) throws PerdixException
        {
                Type t = st.get(name);
                if(t == null)
                        throw new PerdixException("Variable " + name + " not yet declare
d");

                if(indices != null)
                {
                        // type checking for assoc array
                        for (ListIterator iter = indices.listIterator();iter.has
Next();)
                        {
                                Type indexType = ((Expr)iter.next()).typeCheck(s
t);
                                // first the currentType must be an array
                                if (!(t instanceof PerdixArray))
                                        throw new PerdixException ("Trying to index a
non-array type");
                                // second, the type of the index and the key typ
e must match
                                PerdixArray a = (PerdixArray)t;
                                if (a.getKeyType() != indexType)
                                {
                                        throw new PerdixException ("Invalid key type"
);
                                }
                                // everything is checked advance to the next ind
ex
                                t = a.getValueType();
                        }
                }

                return t;
        }

        /* The gen() method of the Variable class should only be used for all ca
ses except
        * for the left hand side of an assignment, and that is why it should alw
ays
        * generate get() and no put()*/
        public String gen()
        {
```



```
                String output=this.name;

                //check if the Variable is an Array
                if (this.indices!=null)
                {
                        for (ListIterator iter = indices.listIterator();iter.has
Next();)
                        {
                                Expr index = (Expr)iter.next();

                                String index_string = "";
                if(index instanceof StringConstant)
                    index_string ="\""+index.gen()+"\"";
                else
                    index_string = index.gen();

                                output += ".get(" +index_string +")" ;
                        }
                }
                return output;
        }
}
```

```
package perdix.core;

import java.util.*;

public class VariableDeclaration extends AbstractExpr{

        String name;
        Type type;
        Expr assign;


        //list of indices instead of simple index
        public VariableDeclaration(String name, Type type, Expr e)
        {
                this.name=name;
                this.type=type;
                this.assign=e;

        }

        public String getName()
        {
                return name;
        }

        public Type getDeclaredType()
        {
                return type;
        }

        private String arrayTypeGen(Type t)
    {
            // if it's an int, return Integer
            //we use "Integer", not "int", because this is inside Java generic
            if (t.equals(Type.PerdixInt)) return "Integer";

            // anything else other than an array will have type String
            if (!(t instanceof PerdixArray)) return "String";

            // if it's an array, downcast it
            PerdixArray a = (PerdixArray)t;
            return "Hashtable< "
                            + arrayTypeGen(a.getKeyType())
                            + ","
                            + arrayTypeGen(a.getValueType()) +
                            " > ";
    }

        public Type typeCheck(SymbolTable st) throws PerdixException
        {

                if((st.get(name))!= null)
                        throw new PerdixException("Variable redeclared!");

                /*check compatibility of rhs and lhs in the initialization*/
                if(assign != null)
                {
                        if (this.type instanceof PerdixArray)
                        {
                                throw new PerdixException("Initialization of arrays is not all
owed in the declaration...!\n");
                        }
                        if((this.type).equals(Type.PerdixInt))
                        {
                                if (!(assign.typeCheck(st)).equals(this.type))
                                        throw new PerdixException("Declared and initia
lized types incompatible!\n");
                        }
```



```
                        /* Variables of type file should be initialized with
                         * other variables of the same type or with strings */
                        if(!((this.type).equals(assign.typeCheck(st)))&&
                                !(((this.type).equals(Type.PerdixFile))&& ((assi
gn.typeCheck(st)).equals(Type.PerdixString))))
                        {

                                throw new PerdixException(this.name+": De
clared and initialized types incompatible!");
                        }

                }

                st.put(name, type); //insert the newly declared variable to the
current scope

                return Type.PerdixVoid;
        }

        public String gen()
        {       String prefix="";
                String middle = "";
                String output="";
                String postfix = "";

                if ((this.type) instanceof PerdixArray)
                {

               String t = arrayTypeGen(this.type);
               output = t + " " + this.name + " = new " + t + "();";

                }
                else
                {
                        /* declare an integer value */
                        if((this.type).equals(Type.PerdixInt))
                        {
                                prefix = "int";
                                if(this.assign != null)
                                        output += "    " + "int " + this.name + " =
" + assign.gen() + ";\n";
                                else
                                        output += "int " + this.name;
                        }
                        else
                                /* otherwise declare a String value */
                                if ((this.type).equals(Type.PerdixFile)|| (this.
type).equals(Type.PerdixIp4address) ||
                                (this.type).equals(Type.PerdixMac) ||(this.type)
.equals(Type.PerdixString))
                                {
                                        prefix = "String";

                                        if(this.assign != null)
                                                output += "    String " + this.name
 + " = " + assign.gen() + ";\n";
                                        else
                                                output += "String " + this.name;
                                }
                        }
                return output;
        }
}
```

**Table of Contents**