Università
Ca'Foscari
Venezia

Master's Degree programme — Second Cycle (*D.M. 270/2004*)
in Informatica — Computer Science

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Final Thesis

A uniform language for firewall configuration
A multi-target compiler for Mignis

Supervisor
Ch. Prof. Riccardo Focardi

Graduand
Alessio Zennaro
Matriculation Number 800194

Academic Year
2015 / 2016

CA' FOSCARI - UNIVERSITY OF VENICE
DEPARTMENT OF ENVIRONMENTAL SCIENCES, INFORMATICS AND STATISTICS

SUPERVISOR:
Chiar.mo Prof. Riccardo Focardi

WRITTEN IN:

Mestre

ABSTRACT

Corporate networks are often complex and can include a big number of firewalls that need to be set up and configured; it is possible that these firewalls are based on different systems and thus different languages to write the rules must be used. This makes the job of a network administrator hard since s/he needs to know a big number of languages to correctly set up and keep updated the network.

Mignis is a semantic based tool for firewall configuration developed by the security group of Ca' Foscari. It provides a simple firewall language that is very easy to learn and use. Unfortunately Mignis is at the moment usable only for Netfilter, Linux firewalls, since its implementation translates the rules using iptables commands.

In this thesis we present a new multi-target compiler for the Mignis language, completely rewritten in order to easily support a translation of Mignis into different target languages; with this approach a network administrator can use a single language to write all the firewall rules of a network (regardless of its complexity) and then compile them into different target languages.

If you spend more on coffee than on IT security, you will be hacked. What's more, you deserve to be hacked.

— Richard A. Clarke - White House Cybersecurity Advisor

ACKNOWLEDGMENTS

I wish to thank professor Riccardo Focardi, who assisted me in the drawing up of this thesis and encouraged me in my passion about computer security.

I owe a special thanks to my former manager, dr. Michela Lazzarini, to my former team leader and to the current one, ing. Fabio Garganego and dr. Piergiorgio Volpago respectively, and to all my colleagues for having facilitated me in every possible way allowing me to attend the lectures.

Another special thanks goes to Claudia, my future sister-in-law, for having helped me write a thesis with a better english.

All my love goes to my girlfriend Giulia, who never made me feel the lack of her full support even when this latter implied a personal sacrifice: for this and for all that she can give me, I want to express to her my deepest gratitude.

Finally, I want to thank with my deepest admiration my parents Monica and Ermanno for having raised me as the man I am and for having always believed in me regardless of my choices.

Desidero ringraziare il Prof. Riccardo Focardi, che mi ha seguito nella stesura di questa tesi e che ha incoraggiato la mia passione per la sicurezza informatica.

Un ringraziamento particolare lo devo alla mia ex dirigente, dr.ssa Michela Lazzarini, al mio ex responsabile e a quello attuale, ing. Fabio Garganego e dr. Piergiorgio Volpago rispettivamente, e a tutti i miei colleghi per avermi in ogni modo agevolato consentendomi di frequentare le lezioni.

Un grazie speciale va a Claudia, la mia futura cognata, per avermi aiutato a scrivere una tesi con un inglese migliore.

Tutto il mio amore va alla mia ragazza Giulia, che non mi ha mai fatto mancare il suo completo supporto anche quando questo le costava un personale sacrificio: per questo e per tutto quello che è in grado di darmi voglio esprimerle la mia più profonda gratitudine.

Infine, voglio ringraziare con profonda ammirazione i miei genitori Monica ed Ermanno per avermi reso l'uomo che sono e per aver sempre creduto in me aldilà delle mie scelte.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Suppose that a college campus has many different buildings and the head of the campus wants to set up a network in which each building has its own network space and for each network space there can be different subnets. In such scenario it is reasonable that all the network spaces are connected with each other through a main firewall with specific rules (for instance, it is possible that the dorms network cannot connect to the labs network). In addition to this, each subnet within a network space could be allowed to connect only to selected subnets (for instance, within the labs network there could be a subnet for each lab and all of them could be allowed to connect to an administrative subnet in which there are shared resources but they could be disallowed to connect with the other labs subnets).

In such scenario we cannot be sure that all the firewalls are implemented using the same hardware or operative system. Each firewall could be based on different systems and thus needs to be set up using its specific language.

The main firewall could be, for example, a big Juniper[1] or Cisco[2] hardware appliance while the smaller firewalls set up in the various buildings could be based on Personal Computers with a Linux distribution as operative system using Netfilter[3] for the rules implementation.

The network administrator must know many different languages

In a situation like the described one, a network administrator must deal with many different languages and to correctly set up all the rules s/he must know these languages very well in order to avoid trivial mistakes.

While a good network administrator must know how to describe network rules at least in theoretical terms, we cannot be sure that s/he knows all the possible different languages used to actually implement them on a device. Sometimes there could be even different dialects of the same language used by the same brand

1 http://www.juniper.net/us/en/products-services/security/
2 http://www.cisco.com/c/en/us/products/security/firewalls/index.html
3 http://www.netfilter.org

based on the line of products (entry level products could use a certain dialect while professional level products could use a slightly different version of the same language to support advanced features).

Mignis is a semantic tool[4] that allows to easily describe firewall rules[1], regardless of the routing. Its main strength is that it uses only four operators and it supports NAT.

Listing 1.1: Example of some mignis rules

```
1 lab_net [.] > internet          # The laboratory net can go to internet
2 dorm_net > service_net          # The dorm network can connect to the service network
3 dorm_net / internet             # The dorm network cannot connecto to internet
```

In listing 1.1 there is an example showing three rules written in Mignis and their meaning is the following: the network *lab_net* is allowed to connect to the internet directly using a masquerade (line 1), the *dorm_net* network is allowed to connect to *service_net* (line 2) while the same *dorm_net* is not allowed to connect to internet directly (line 3). This is just a fragment of the whole configuration, since, in order to work, there should be an explicit definition of the interfaces and the aliases (i.e. *dorm_net*, *internet* and *service_net*).

The grammar and the structure of a Mignis configuration file is better explained in chapter 4

Mignis *simplifies the use of iptables*

Since its first version until the current official one, the Mignis compiler translates the firewall rules written in Mignis only into a Netfilter based configuration and so we can think of Mignis also (but not only) as a way to simplify the rather complex structure of a configuration written using iptables.

It is not difficult to imagine, however, that it is enough to rewrite the current compiler in order to support more target languages other than iptables in order to be able to use Mignis with the goal to write, for example, all the rules the network administrator has to implement, possibly on different devices, using a single language. Eventually the compiling process will have to produce the output using different target languages, depending on which are the needs.

In the meantime, Mignis has been expanded (at least in theoretical terms) in order to support more expressive rules with the goal of being able to localize firewall

4 https://github.com/secgroup/Mignis

security policies[2]. This is achived by slightly changing its original grammar, making possible to explicitly state which interface we are expecting the packets to come from, or be sent to. This expansion of the original Mignis language is called *Mignis+*.

Mignis+ *allows for more expressive rules*

Listing 1.2 shows an example of how a rule written using Mignis+ looks like.

Listing 1.2: Example of Mignis+

```
1 # Computer Science Laboratory network can access the shared resources server at port
      443
2 cs_lab@If_CS_Lab > res_server@If_RES:443 tcp
```

Provided that *cs_lab*, *If_CS_Lab*, *res_server* and *If_RES* are correctly defined, this rule simply states that the cs_lab network can connect to a server (identified by res_server) at port 443 and using the TCP protocol as long as the source packets come from the If_CS_Lab interface and the destination packets go to the If_Res interface.

The final target languages can be very different from each other and the simple structure of a Mignis(+) configuration file is not really fit to be compiled directly to the selected target language: the best option is to perform the lexical and syntactical analysis using a compiler that produces a machine-readable configuration file written in a "low level" intermediate language in which all the potentially needed information are represented (possibly with an empty field); a translator can then loop on all the rows produced in the previous step and perform the translation into the target language.

In this document the compiler that produces the intermediate language is often called the *backend component* while the part that takes the intermediate language as input and produces a certain target language as output is often called the *frontend component*; a configuration written in the intermediate language is often called *intermediate representation*.

In this thesis the new multi-target compiler of Mignis(+) is designed and presented: in chapter 2 Mignis and Mignis+ are briefly presented, some theoretical concepts about the lexical analysis and the parsing of a language are recalled and, finally, the description of the tools that will be used to implement the compiler are described; in chapter 3 the intermediate language is presented, defined and

described; in chapter 4 the structure of a Mignis(+) configuration file is analized in order to create a lexer, a parser and, eventually, a real compiler (backend component); in chapter 5 the translation process towards a target languages is discussed (frontend component) and eventually, in chapter 6, there are the conclusions and the description of the possible future works together with the known limitation of the what has been done up to now.

2

THEORETICAL REFERENCES AND PREREQUISITES

Writing a new compiler is a complex task and before being ready to deeply analyze all the lexical and grammatical details, it is important to have a clear view of how the language to be compiled actually looks like.

This helps to avoid trivial mistakes while implementing the core parts of the backend component and perform a correct translation when writing the frontend component.

Moreover, it is really important to describe the tools that are going to be used to write the new compiler along with all the underlying theoretical concepts.

In this chapter, Mignis and Mignis+ are briefly presented: the main parts of a configuration files are described and the meaning of operators is explained. A deeper analysis of the Mignis(+) syntax and structure can be found in chapter 4.

After that, some theoretical concepts about lexical analysis and parsing are discussed, together with a very brief description of the tools, provided by OCaml, used to actually implement the backend component.

2.1 MIGNIS AND MIGNIS+: A BRIEF DESCRIPTION

Mignis and Mignis+ are tools used to configure firewalls. When dealing with network security, firewalls are an essential component because they are capable of filtering packets.

Mignis, presented in 2014, is very simple to learn and use and allows for non-trivial configurations including advanced features such as NAT[1].

However, Mignis does not allow for policy localization even if such a thing could be very useful in complex networks. In 2016, Mignis+ has been presented and it allows to localize security policies by extending the original Mignis or imposing some restrictions[2].

In this section we briefly describe both Mignis and Mignis+, in order to have a better knowledge of what we are working on.

2.1.1 *Mignis*

A Mignis configuration file consists of a set of six sections:

1. *Section* ***OPTIONS***: a useful part in which options can be set on or off (e.g., logging);
2. *Section* ***INTERFACES***: in this part physical interfaces (e.g., `eth0`) are bound to network addresses (in the standard form `IP/n` where `n` is the number of bits set to 1 in the subnet mask, starting left) and a name is given to this bound;
3. *Section* ***ALIASES***: here host IP or network IP addresses are given a name in order to make the configuration more human readable;
4. *Section* ***FIREWALL***: this is the main section, in which all the rules are described using the Mignis operators;
5. *Section* ***POLICIES***: default rules can be specified here. Since they are default rules, they are matched only if a packet does not match any other rule stated in the FIREWALL section;
6. *Section* ***CUSTOM***: rules written directly in a target language can be added[1] in this section.

Comments are allowed and they are always *inline* comments, meaning that there is a special character that marks the beginning of the comment and it ends when a new line character is found on that line.

The special character that marks the beginning of a comment is the *sharp* (#) character.

In listing 2.1 it is shown a typical Mignis configuration file.

1 The current official version of Mignis supports only iptables commands

Listing 2.1: An example of Mignis configuration file

```
1  OPTIONS
2  default_rules   no
3  logging         no
4  established     yes
5
6  INTERFACES
7  wan             eth2 0.0.0.0/0          # Internet
8  lan             eth0 10.0.0.0/8         # Cabled
9  wlan            eth1 172.22.0.0/16      # Wireless
10
11 ALIASES
12 mypc            10.0.0.2                # My pc
13 router          1.2.3.4                 # External router address
14 server          10.0.0.3                # An internal server
15 mal             192.168.1.0/24          # A malicious network
16
17 FIREWALL
18 lan [.] >       wan                     # lan can go outside with masquerade
19 wlan    /       wan                     # wlan cannot go outside
20 wlan    >       mypc : 8080 tcp         # wlan can connect to my pc at port 8080 (tcp)
21 *       >       [router:80] server:80   # Anything can connect to server:80 with dNAT
22 *       /       mal                     # Nothing chan connect to the mal network
23
24 POLICIES
25 *       /       lan
26
27 CUSTOM
28 -A INPUT -p tcp --dport 7792 -j LOG --log-prefix "PORT 7792 "
29 -A INPUT -p tcp --dport 7792 -j ACCEPT
```

It is important to spend a few words about the *FIREWALL* section, at least to describe the meaning of the various operators. Basically this section is made by a set of rules and each rule is made by five main parts:

1. A source node, which can be a network or host IP address, an alias or an interface name. A source NAT (sNAT), including a masquerade, may be specified (see line 18 in listing 2.1);

2. A destination node, with the same features of the source nodes. A destination NAT (dNAT) may be specified (see line 21 in listing 2.1);

3. An operator, put between the source and destination nodes, that is used to express whether a packet is allowed to flow from the source to the destination or not;

4. A protocol, if the rule is matched only when a certain protocol is used. This part is optional (shown at line 20 in listing 2.1);

5. An additional optional rule part, written directly in the target language, separated from the rest by a pipe (|) character (not shown in listing 2.1).

Port filtering is allowed

When writing the endpoints of the communication (i.e. source and destination nodes), it is possible to also add a specific port in order to restrict the rule effect only to that port. This can be useful, for instance, when a packet is allowed to flow from the source node to a destination node only when a specific port is used, while the use of any other port is forbidden for the same packet with the same source and destination; this situation is very common in real life (e.g., a webserver that is willing to respond only to requests made to port 80).

Each rule must use an operator and the Mignis language provides four of them:

- *Operator* > : This operator is used to **allow** a packet to flow from the source node to the destination node but *not* vice versa;
- *Operator* <> : This operator is used to **allow** a packet to flow from the source node to the destionation node *and* vice versa;
- *Operator* / : This operator is used to **forbid** a packet to flow from the source node to the destination node without giving any response to the source node (*DROP*);
- *Operator* // : This operator is used to **forbid** a packet to flow from the source node to the destination node but a "destination-unreachable" ICMP message is sent to the source node (*REJECT*).

NAT endpoints are written between square brackets, while masquerade is denoted by a dot (.) character between square brackets.

Mignis, at the moment, is a Python program made of about 1500 code lines[1] that, taken a set of rules written in the language informally just described, returns a list of Netfilter/iptables rules that can be applied on a real life environment.

Mignis checks important conditions

The tool checks some conditions: NAT-safety, NAT-consistency, no local sNAT rules and Determinism[1].

The new compiler must check these conditions as well, even if its goal is not to specifically translate into Netfilter/iptables but to, potentially, every real firewall

language. This means that some little changes to the original Mignis can be made, but these changes must not compromise the global correctness of the original tool.

2.1.2 *Mignis+*

Mignis is useful if there is only one firewall to set up (e.g., for a client computer). but when it comes to something more complex like setting up the rules for a whole corporate network, then it shows some limitations.

Suppose a situation similar to the one shown in Figure 1[2] in which it is rather reasonable that we want to configure, among others, these particular rules:

1. A machine in the *Trusted* segment is allowed to connect to a machine in the *Sensitive* segment when packets flow from *Trusted* to *fw2*, then to *fw1* and eventually to *Sensitive*;
2. The same machine in the *Trusted* segment is **not** allowed to connect to a machine in the *Sensitive* segment when packets flow from *Trusted* to *fw2*, then to *fw4*, then to the internet and then, somehow, to *fw3*, then to *fw1* and eventually to *Sensitive*.

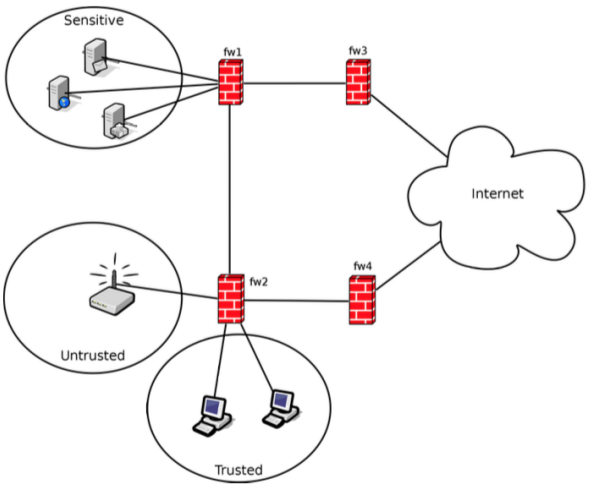

Figure 1: Example of a complete network with Trusted, Sensitive and Untrusted segments

2 Image taken from [2]

In order to satisfy rule number 1, packets must be allowed to flow through *fw2* and *fw1* but, in order to satisfy rule number 2, it is essential to be able to explicitly specify which is the interface from which the packet is allowed to come from (or sent to).

In other words, we want to be able to say: a packet p, with source in the *Trusted* network and destination in the *Sensitive* network, is allowed to flow through firewall *fw1* only if it comes from the network interface to which is connected firewall *fw2*. If the same packet comes from the interface to which is connected firewall *fw3*, then drop it.

Mignis+ extends Mignis and has some extra features

Mignis+ is an extension of Mignis and it has these main features/restrictions[2]:

- It must be possible to localize a rule in order to filter packets depending on the network topology;
- Established connection are not granted by default;
- Rules are all positive, which means that only the operators > and <> are allowed to be used in a Mignis+ configuration.

A fragment of a Mignis+ configuration is shown in listing 2.2.

Listing 2.2: A little fragment of a Mignis+ configuration file

```
1  ...
2  INTERFACES
3  if_fw3          eth0    0.0.0.0/0
4  if_Sens         eth1    10.0.0.0/24
5  if_fw2          eth2    0.0.0.0/0
6
7  ALIASES
8  Sensitive       10.0.0.0/24
9  Untrusted       10.1.1.0/24
10 Trusted         10.1.2.0/24
11 Internet        0.0.0.0/0
12
13 FIREWALL
14 Sensitive@if_Sens:22    >       Trusted@if_fw2  | -m state --state ESTABLISHED,RELATED
15 Sensitive@if_Sens       >       Internet@if_fw2:443
16
17 Sensitive@if_Sens       >       Internet@if_fw3:443
18 ...
```

As it is easy to notice by looking at lines 14, 15 and 17 in listing 2.2, the localization of a rule is done by putting an at (@) character immediately after the

host and specifying (immediately after the at) the interface the packet must come from (or be forwarded to) in order to match the rule.

2.2 LEXICAL ANALYSIS

The goal of a lexical analyzer is to transform a generic sequence of characters into a sequence of "words" provided by the lexicon of the language. More formally, all the language's lexemes must be recognized in order to return all the associated tokens.

Normally this is done by defining the patterns that describe how the lexeme of a token "looks like".

It is useful to discuss some theoretical concepts, before actually present a tool that performs the analysis described in 4.1.

2.2.1 *Concepts about lexical analysis*

When dealing with lexical analysis, we are requested to solve two main problems that can be summarized as follows:

1. Being able to correctly describe lexemes in order to recognize them and return the correct associated tokens; the recognition process must be unambiguous. A second aspect of this task is to find and use efficient algorithms capable to actually recognize a lexeme described by a pattern;

2. Being able to isolate groups of characters from the input stream that form a formally described lexeme.

It is enough to use automata theory to obtain powerful tools that can be employed to solve the problems addressed at point 1.

Regular expressions and DFA*s are powerful tools*

Indeed, patterns are basically regular sets that can be formalized using regular expressions[3]; since regular expressions can be easily recognized by using deterministic finite automata (*DFA*)[4], we can also use these latter to recognize lexemes.

Point 2 is a little bit more tricky: at first the solution could be scanning the input flow one character at a time and use the next character to find which class of lexemes we are currently scanning. We should use the same approach to decide whether a certain lexeme is finished or not.

While this approach is very intuitive and easy to implement, several problems arise:

1. **Common prefixes**: this problem arises when two or more different lexemes share a common prefix. When this happens there could be a string that could match a regular expression in more than one way, possibly returning a different sequence of tokens depending on which rule is considered the one to be matched. This is an ambiguity and thus it cannot be tolerated;

2. **Overlapping lexemes**: sometimes there are lexemes that completely overlap, meaning that they look identical but they are associated with different tokens. This is the case of an identifier and a keyword that could look exactly the same but their meaning is completely different;

To solve problem 1 we must decide which rule "wins" and the most logical choice is to apply the "Longest Match Rule": whenever there is a common prefix, the recognition process will not stop until there are no shared prefixes anymore and thus returning the smallest possible number of tokens.

Longest Match Rule introduces another problem

This approach leads to another problem: if we find in the input stream a sequence that could be recognized by two or more rules, we try to recognize the longest one. At the moment we decide to go on, we cannot be sure that this approach will succeed but, at the same time, we cannot fail without having tried to recognize smaller tokens. In order to be able to perform such a thing, proper rollback policies must be implemented in order to fail only when no rule can be matched.

Problem number 2 is very easy to solve: it is enough to decide priorities. We can, for instance, decide that a language keyword has a higher priority than an identifier and so a string with a keyword is always recognized as a keyword token and never as an identifier token. This implicitly causes the impossibility to use a keyword as an identifier, because this would generate a parsing error (there would be a keyword token when an identifier one is expected).

2.2.2 *OCamllex*

OCamllex is a tool that is used to produce a lexical analyzer starting from a set of regular expressions with specific semantic actions. The result is an OCaml program that is able to recognize the strings that match against the regular expressions and return a sequence of tokens[5].

An OCamllex definition has this fundamental structure, as shown in listing 2.3[3]:

Listing 2.3: Structure of an OCamllex definition

```
1  {
2          header
3  }
4
5  let abbreviation = RegExp
6
7  rule EntryPoint_1 = parse
8    | RegExp_1    { Action_1 }
9    | RegExp_2    { Action_2 }
10   | ...
11   | RegExp_n    { Action_n }
12 and EntryPoint_2 = parse
13   | ...
14 and ...
15 {
16         trailer
17 }
```

In the header (lines 1-3) and in the trailer (lines 15-17) we can put auxiliary OCaml functions (that can be used during the lexing) or type definitions; modules can be opened as well in the header.

In line 5 we introduce an identifier that is used to give a name to a regular expression, particularly useful when this latter is rather long. From this point of view, we are allowed to introduce abbreviations.

Lines 7-14 are the very heart of an OCamllex definition: each entry point is essentially an OCaml function that wants at least one argument of type *Lexing.lexbuf*, which is the input stream we want to perform the lexical analysis on; more optional parameters may be specified. The main input is matched against all the regular expressions written in the rule until one of them is actually matched: if this happens the corresponding action is taken and the result is returned by the

3 The presented structure is simplified with respect to the one described in the official manual

function represented by the entry point. The Longest Matching Rule is applied in case of overlapping rules; if rules still overlap, then the one that comes earlier in the definition wins (see 2.2.1).

The regular expressions must be written with a specific syntax which is rather intuitive but sometimes a little peculiar. The main features are:

- A character must be put between single quotes (') while strings must be put between double quotes (");
- To match any character we can use the wild card _ ;
- A character belonging to a set is denoted with [char-set] where *char-set* can be a range ($'c_1' - 'c_2'$), a single character (c) or an union of character sets;
- A charcter which does not belong to a set of character is denoted with [^ char-set] where *char-set* is the same as the previous point;
- To match a concatenation of zero or more repetition of a certain regular expression it is enough to put a star character (*) after the regular expression, while to match a concatenation of at least one repetition of a regular expression the plus character (+) is to be used;
- To match something that could not appear, or could appear only once, it is enough to use the question mark (?);
- If two or more regular expressions are alternative, meaning that one of them has to be matched, it is enough to use the pipe character (|) between the various regular expressions.

To bind a string that matches a regular expression to an identifier it is enough to write the regular expression followed by the keyword *as* and an identifier: by doing so, the identifier can be used in the associated action to manipulate the string that matched against the regular expression (e.g., $\mathtt{RegExp}_n$ as $variable\_name$).

2.3 PARSING

The goal of the lexical analysis discussed so far (see 2.2) is to transform a stream of character into a stream of tokens. When the lexical analysis is over, we know

that everything that is written in the input file belongs to the language lexicon. The next step is to check whether the token stream returned in the previous step complies with the formal grammar of the language. In other words the next step is to verify that the tokens are not put at random, but they follow an order defined by the syntax of the language.

This is the goal of the parsing phase: the stream of tokens representing the language lexicon is checked and if it complies with the formal grammar, an Abstract Syntax Tree (AST) is returned.

2.3.1 *Parsing concepts and context-free grammars*

While regular expressions were the perfect tool to be used for the description of the patterns that represent the lexemes associated with the language tokens, they are definitely not suitable to describe the grammar of a language.

Languages cannot be described with regular expressions

This is because programming languages uses a lot of balanced items (e.g., parenthesis) and this class of constructs cannot be described by a regular expression (so they are not regular languages) and thus cannot be recognized by a DFA.

Indeed, it can be easily shown that a language described as $L = \{0^n\,1^n \,|n \geqslant 1\}$[4] is not regular and thus it is not possible to describe it with a regular expression (and, as a consequence of this, there is no finite automata able to recognize it)[6]; since the discussion about regular languages is not the goal of this thesis, the formal proof is left to the reader.

However, it is easy to notice that L is the typical language with balanced items: it is the language with n 0s, followed by n 1s. 0s and 1s can be thought as parenthesis.

Programming languages are defined using *context-free grammars*. A context-free grammar can be formally defined as a 4-tuple:

$$\mathfrak{G} = (\mathcal{N}, \Sigma, \rightarrow, \mathcal{S})$$

where

- $\mathcal{N}$ is the set of non-terminals symbols;

4 c_1^i has to be interpreted as "the character c_1 is repeated i times"

- Σ is the set of terminal symbols, also known as the *alphabet*
- $\rightarrow$ is the set of production rules in the form $\mathcal{N} \rightarrow (\Sigma \cup \mathcal{N})^*$;
- $\mathcal{S} \in \mathcal{N}$ is a non-terminal symbol that is used as *starting symbol*.

All the sets must be finite.

More informally a context-free grammar has a certain number of non-terminal symbols and it produces strings of terminal symbols belonging to an alphabet. The production rules say how *one* non-terminal can be expanded in a sequence of terminals and non-terminals.

Given a grammar $\mathfrak{G}$, the set of terminal strings that can be derived from the starting symbol $\mathcal{S}$ is the context-free language defined by $\mathfrak{G}$; each of these strings are called a *sentence*.

Parsing an input string means to try to find a derivation that from the starting symbol $\mathcal{S}$ can eventually arrive to the input string itself (obviously using the production rules of the grammar).

The most intuitive way to parse an input string is called *top-down parsing*: given an input string to parse, we begin from the starting symbol and we try to build the *parse tree* by expanding the non-terminal symbols until we are able to arrive to the input string or fail. Despite being very easy to understand, this kind of parsing algorithm is very inefficient.

To increase efficiency we use a predictive parsing table

In order to enhance efficiency, we can use what is called *predictive parsing*: we have to use a so-called "parsing table" that is a two dimensional matrix in which the columns are all the terminal symbols and the rows are all the non-terminal symbols. Each couple (A, c) (where A is a non-terminal symbol and c is a terminal symbol) is associated a production rule $r \in \rightarrow$ or an error code. Intuitively at each step the parser tries to expand the non-terminal symbols by using the predictive parsing table that tells which production rule has to be used given the current non-terminal to be expanded and the current character from the input string. If no rule can be applied, than the parser fails; if the input string ends and at each step a production has been successfully applied, then the parsing succeeded.

There are two main problems that could arise when dealing with this very intuitive approach:

1. There are grammars that produce parsing table in which for some entries there are more than one production rule, and this introduces ambiguities;
2. When a grammar is *left recursive*[5], intuitively the parser could enter an infinite loop.

When none of these problems actually arise, then the grammar is called *LL(1)*.

While there are ways and algorithms to manipulate the grammars in order to eliminate ambiguities and left recursion, we do not have any assurance that the final result will be a LL(1) grammar.

There is a much more powerful parsing technique that is called *bottom-up parsing* in which we start from the input string (the one to be parsed) and then we try to arrive to the starting symbol applying the production rules but not by expanding them but by reducing them. This technique is based on another class of grammars, called *LR* grammars, which is more expressive then LL(1): left recursive grammars, for example, can be LR grammars. However, ambiguity could be still a problem.

Bottom-up parsing is more powerful than top-down parsing and for this reason it is the most used technique in the modern parsers. However, it is also much more complex and less intuitive.

The Mignis(+) parser, discussed in 4.2.3, is indeed a bottom-up one because the used tool generates this kind of parsers.

These are few information about parsers and the reader that finds this topic interesting is invited to read [3, 7], from which these theoretical references have been taken.

2.3.2 *OCamlyacc*

OCamlyacc is a tool that is used to produce a parser starting from a context-free grammar specification with semantic actions attached to it (just like yacc). The result is the OCaml code for the parser and basically the produced module defines an OCaml function for each entry point. These are the parsing functions[5].

5 A grammar is left recursive when it contains at least one non-terminal symbol that, directly or indirectly, appears again after having being expanded in the leftmost position

A typical OCamlyacc defintion has the structure shown in listing 2.4[6].

Listing 2.4: Structure of an OCamlyacc definition

```
1  %{
2          header
3  %}
4  /* declarations */
5  %token TK1 TK2 ...
6  %token <type_expr> TKN TKM ...
7
8  %start entry_point_symbol
9  %type <type_expr> entry_point_function
10
11 %%
12 /* rules */
13 nonterminal_symbol:
14   Symbol ... Symbol     { semantic_action_1 }
15  |Symbol ... Symbol     { semantic_action_2 }
16  | ...                  { ... }
17 ;
18 entry_point_symbol:
19   Symbol ... Symbol     { semantic_action_ep }
20  | ...                  { ... }
21 ;
22
23 %%
24 trailer
```

At lines 1-3 there is the header and, within it, it is possible to write custom functions useful for the data manipulations and it is also possible to open modules (if they are needed).

Lines 4-10 is where the *declarations* are put. There are many things that can be declared here but three of them are particularly interesting:

- A sequence of *%token* keywords followed by the name of the tokens produced by the lexer. All the tokens are separated by a white space as shown at line 5. It is possible to specify a type associated to a token if that token takes with it more information. For example an IDENTIFIER token is always together with the string representing the identifier itself (this is shown at line 6). We can think of this sequence of tokens as the actual definition of the set Σ in the formal grammar;

- One or more *%start* keywords (line 8) defining the entry points for the parser. Basically here we define the start symbol $\mathcal{S}$ of the formal grammar. This is a mandatory part: we are requested to specify which is the start symbols and

6 This structure is simplified since all the features provided by OCamlyacc are not required for the Mignis(+) parser discusse in 4.2.3

they must be non-terminals for which a rule has to be defined in the section dedicated to the rules;

- One or more *%type* keywords (line 9) defining the type that must be associated to a non-terminal symbol. It is mandatory only for those non-terminal symbols that are also declared as start symbols.

At lines 11-21 there is the very heart of an OCamlyacc definition: the rules section.

Within this section all the non-terminal symbols are formally described (according to the context-free grammar that is being implemented) as a sequence of terminals and non-terminals. To each of these sequences is associated a semantic action that must be taken when the pattern describing that rule is matched. In other workds, this section is the one that formally implements the set of production rules $\rightarrow$ and, implicitly, where the set of non-terminal symbols $\mathcal{N}$ is defined.

As already stated, at least one of the non-terminal symbols must be declared as the start symbol and a type must be associated to it. It is important to point up that OCamlyacc produces an OCaml function only for the non-terminals declared as start symbols and the name of the start symbols will be the name of the functions as well. All the other non-terminals are implemented but not as independent functions. This means that the parser will recognize things only starting from the definition associated to a start symbol (which is coherent to the meaning of start symbol in a context-free grammar).

OCamlyacc produces parsing functions with the name of the start symbols

These so-called parsing functions take two arguments: a lexical analyzer, namely a function that transform a lexer buffer into a stream of tokens, and an actual lexer buffer. This means that the lexer is used directly by the parser and it is not to be called independently, it is enough to pass to the parser the entry point for the lexer. Each parsing function returns a semantic attribute that must be of the type declared with the *%type* declaration for the start symbol that generated the current parsing function, as discussed above.

Eventually, at lines 23-24 there is the trailing section in which other auxiliary functions can be defined as well.

3

REPRESENTING MIGNIS AND MIGNIS+ AT A LOWER LEVEL: THE INTERMEDIATE LANGUAGE

The backend component described in chapter 4 takes as input a file in which a Mignis(+) configuration has been written and produces as output as many files as the number of firewall configurations found in the input. These output files are not written directly in the target language: the goal of this new compiler is to make Mignis(+) a tool able to produce firewall configurations regardless of the actual used system or hardware.

The current version of Mignis produces, as already stated, only Netfilter configurations written with the use of the iptables command. The compiler creates the Netfilters rules directly from the Mignis representation of the rules[1], but this is an approach that is not really fit to allow for the support of different target languages.

In this chapter we present an intermediate language that is used to represent Mignis(+) in an indeed *intermediate representation*; we first discuss the reasons of this choice, then the language itself is formally introduced with the use of a context-free grammar and eventually we also describe how this language is used in order to write Mignis(+) equivalent rules.

3.1 THE NEED OF AN INTERMEDIATE LANGUAGE

Providing a tool that must be used on different platforms or system is not a new problem in Computer Science. Think of the Java programming language: it has been created to be cross-platform, which means that it is hardware and operative system independent[8].

From a certain point of view, the task of transforming Mignis(+) from a Netfilter related tool to an independent instrument that can be used regardless of the actual

hardware or operative system that will enforce the rules, is not so different from the Java example.

In both cases the final goal is to have a tool that works without the concern of knowing which will be the machine that will execute the code (for Java) or the firewall that will forward or drop the packets in a network (for Mignis).

On the other hand, the biggest and most relevant difference is that Java is a programming language and its code must be executed by a machine, while Mignis(+) is just a tool to formalize firewall rules that will be set on a device.

Java allows a programmer to write the source code and then the compiler produces a *bytecode* that must be executed by a virtual machine (called "*Java Virtual Machine*") and this bytecode, at least theoretically, can be executed on whatever platform without having the need to recompile it (as long as a proper virtual machine exists for that platform). The original source code is not directly interpreted by the virtual machine because of efficiency reasons; it would also be particularly complicated to interpret on the fly a high level language with all its syntactical structures and sugars.

Using a lower level intermediate language is convenient and enhance efficiency

The approach that we used when developing the new Mignis(+) compiler is, at some extent, very similar and it is based on the same theoretical principle: it is convenient to produce a lower level intermediate language that is independent from the actual hardware or operative system and it is therefore easier to manage and use: being at a lower level, an intermediate language has simpler structures and must represent also the information that are not directly stated in the original language but needed to correctly work on an actual device.

Also, the aliases are resolved, so symbolic names are not allowed to appear in a intermediate language written rule, making this representation less complex.

Globally, using a lower level intermediate language can help to enhance the overall efficiency.

However, it is not so difficult to notice that this approach alone is completely and deeply wrong: we do not have any virtual machine that can run on firewalls hardware interpreting the "Mignis(+) bytecode". When rules are passed to the actual devices, they must be written in the specific language the device is able to understand.

While Java is never compiled in a "target machine code", Mignis(+) final output must be a list of rules written in a pure target language[1].

What presented so far in this section is then applied with a different goal (with respect to Java): make the final production of the list of rules to be written in the target language much more easier.

Since the intermediate language is much simpler and has a fundamental low level structure, it is possible to run a smaller program (compared to a complete compiler) that simply translate the rules written in the intermediate language into rules written in a target language. This latter program must be able to perform the translation with multiple target languages. This also makes the final translation independent from the lexical analysis and the parsing.

Every rule written in the intermediate language has to be produced by the backend component described in chapter 4 and so we can assume that there are no syntax error. This is a strong assumption and in order to be fulfilled it is strongly reccomanded to never write rules in the intermediate language by hand. The intermediate language should be used to write rules only by the backend component.

The intermediate language is never to be used to write rules by hand

To sum up, this "Mignis(+) bytecode" has been created (and was necessary) to make the translation towards a target language independent from the complexity of the high level Mignis(+) and to enhance the overall efficiency of the translation process that is now completely unrelated from the lexical analysis and parsing of the Mignis(+) language.

The translation process and the description of the frontend component that performs it is described in details in chapter 5

3.2 DEFINING THE INTERMEDIATE LANGUAGE

Before actually implementing the backend component, it is essential to provide a formal definition of the intermediate language that will be the output of the compiler and the input of the final translator which will be the frontend component (see 5).

1 By "pure" it is meant that no Mignis(+) elements are allowed

We first provide the grammar of the intermediate language, expressed, as it has been explained in 2.3.1, in form of a context-free grammar. This allows us to have a clear idea of how Mignis(+) rules written by the user will be represented before being actually translated into a sequence of rules to be issued to a real life firewall.

But the formal grammar is not enough to have a full understanding of the meaning of each element in the language: for this reason, we must provide a complete explanation of the structure of each element in the intermediate language which is essential in order to be able to write a frontend component that correctly perform the translation into the target language.

3.2.1 *Formal grammar*

Even though the intermediate language will not be compiled but simply translated into the chosen target language, it needs to be formally defined anyway.

Since we are talking about another language, it is convenient to use a context-free grammar, given that we have already discussed how to define one in 2.3.1.

Before giving the definition of all the sets needed to create a context-free grammar, it is important to stress that, unlike the grammar shown in 4.2.1, the terminal symbols are not going to be represented with tokens returned by a lexical analysis. This latter phase of a compiling process is completely absent and the grammar defined in this section is much more theoretical since it is used to formally describe the intermediate language but it is not implemented in a parser.

The intermediate language is assumed to be used correctly and this is never checked

As already stated, the syntactical and lexical correctness of a configuration written with the intermediate language is assumed to be verified and there is no formal check of this. The intermediate language is indeed translated and *not* compiled. The translation process simply returns unpredictable results (both exceptions and/or wrong final rules) if the grammar here described is not matched.

First of all we need to define the set of terminal symbols Σ_I as follows:

$$\begin{aligned}\Sigma_I = \{&'\text{OPTN}','\text{BIND}','\text{DROP}','\text{RJCT}','\text{ALLW}','\text{TLLW}','\text{PDRP}','\text{PRJC}',\\ &'\text{CSTM}','\text{ANY}','\text{LOCAL}','\text{MASQUERADE}','\text{TCP}','\text{UDP}',\\ &'\text{ICMP}','\text{n}-','\text{h}-','.',';',':','/','\backslash\text{n}', \text{String}, \text{Int}\}\end{aligned}$$

Since the terminal symbols are not the set of tokens returned by a lexer, in the set Σ_I are written directly the strings (between single quotes) that are supposed to be found in the language with two exception: `String` and `Int`. These two are not direct terminal symbols but are the name of two regular expressions that have been used for the sake of clarity. The following are the definitions of such regular expressions using the same conventions used in 4.1.1[2].

$$[A-Za-z0-9 :space: -\ \_\ ']^{+} \Longrightarrow \texttt{String} \tag{3.1}$$

$$[0-9]^{+} \Longrightarrow \texttt{Int} \tag{3.2}$$

It is not difficult to notice that the `String` regular expression is a name for a sequence (of length at least one) of upper and lower case characters, numbers, spaces and other few symbols. `Int` is the name for a sequence of at least one number.

The following is the definition of the set $\mathcal{N}_I$ of non-terminal symbols:

$$\begin{aligned}\mathcal{N}_I = \{&\texttt{Conf}, \texttt{Option\_List}, \texttt{Binding\_List}, \texttt{Rule\_List}, \texttt{Policy\_List},\\ &\texttt{Custom\_List}, \texttt{Endpoint}, \texttt{Protocol}, \texttt{Host}, \texttt{Interface}, \texttt{Net\_IP},\\ &\texttt{Simple\_IP}, \texttt{Addr}\}\end{aligned}$$

And, eventually, the following is the definition of the set of production rules $\rightarrow_I$. They are not shown in a form of set but as a list of rules; the conventions and notation abuses are the same as in section 4.2.1[3]:

$$\begin{aligned}\texttt{conf} \rightarrow\ &\texttt{Option\_List Binding\_List Rule\_List Policy\_List}\\ &\texttt{Custom\_List}\end{aligned} \tag{3.3}$$

$$\texttt{Option\_List} \rightarrow (\texttt{'OPTN' ':' String ';' String '\textbackslash n'})^{*} \tag{3.4}$$

$$\texttt{Binding\_List} \rightarrow (\texttt{'BIND' ':' String ';' Net\_IP '\textbackslash n'})^{*} \tag{3.5}$$

2 These conventions are not explained in this chapter because they are much more related to the regular expressions used to describe patterns in the lexer; in this context, we used the same conventions in order to make this document more uniform

3 Same as note 2 but for CFG

$$
\begin{aligned}
\text{Rule\_List} \rightarrow & ([\text{'DROP'} \mid \text{'RJCT'} \mid \text{'ALLW'} \mid \text{'TLLW'}]\ \text{':'} \\
& \text{Endpoint ';' Endpoint ';' Endpoint ';' Endpoint ';'} \\
& \text{Protocol ';' [String} \mid \epsilon\ ]\ \text{'\textbackslash n'})^* && (3.6)\\
\text{Policy\_List} \rightarrow & ([\text{'PDRP'} \mid \text{'PRJC'}]\ \text{':' Endpoint ';' Endpoint ';'} \\
& \text{Protocol '\textbackslash n'})^* && (3.7)\\
\text{Custom\_List} \rightarrow & (\text{'CSTM' ':' String '\textbackslash n'})^* && (3.8)\\
\text{Endpoint} \rightarrow & \text{ Host ';' Interface ';' Int} && (3.9)\\
\text{Protocol} \rightarrow & \text{ 'TCP'} \\
& |\text{'UDP'} \\
& |\text{'ICMP'} \\
& |\text{'ANY'} && (3.10)\\
\text{Host} \rightarrow & \text{ String} \\
& |\text{Addr} \\
& |\text{'ANY'} \\
& |\text{'LOCAL'} \\
& |\text{'MASQUERADE'} \\
& |\epsilon && (3.11)\\
\text{Interface} \rightarrow & \text{ String} \\
& |\epsilon && (3.12)\\
\text{Simple\_IP} \rightarrow & \text{ Int '.' Int '.' Int '.' Int} && (3.13)\\
\text{Net\_IP} \rightarrow & \text{ Simple\_IP '/' Int} && (3.14)\\
\text{Addr} \rightarrow & \text{ 'h-' Simple\_IP} \\
& |\text{'n-' Net\_IP} && (3.15)
\end{aligned}
$$

It is rather easy to notice that a configuration written in intermediate language is nothing more than a sequence of strings separated by newline characters. Each line has its own grammar and meaning (see 3.2.2).

In order to formally define the grammar of the intermediate language, we need to define the starting symbol $\mathcal{S}_I$ which is the non-terminal conf.

Now that all the sets are defined, we can say that the grammar:

$$\mathfrak{G}_I = (\mathcal{N}_I, \Sigma_I, \rightarrow_I, \mathcal{S}_I)$$

formally describes the intermediate language that represents at a lower level a Mignis(+) configuration after that this latter has been compiled by the backend component described in chapter 4.

The final compiler discussed in 4.3.2 produces a set of files which are written in a language that complies with $\mathfrak{G}_I$.

3.2.2 *Rules structure and meaning*

So far we have presented the formal grammar of the intermediate language but nothing have been said about the meaning of the statements that can be found. Basically a configuration file written in the Mignis(+) intermediate language is a sequence of statements separated by a newline (`\n`) character.

The intermediate language has been thought to be easily "understood" by a machine. Reading a configuration file written in the intermediate language is not particularly easy for the human being, but its understanding is essential in order to create a final translator that correctly do its job.

Basically, each line can express a rule belonging to one of the following categories:

1. **Options**: rules belonging to this category begin with the keyword *OPTN* and are the only ones that are not supposed to be translated directly into rules written in the target language. Instead, they activate (or deactivate) special features in the final translator;

2. **Interface bindings**: rules belonging to this category begin with the keyword *BIND* and they are used to bind a physical interface to a network IP address. It depends on the final target language whether they are translated into an actual rule or not; however, these bindings can be used also to resolve an interface's physical name (e.g., `eth0`) to a network IP address;

3. **Rules**: rules belonging to this category can be further split into subcategories:

    a) **Firewall rules**: they begin with keywords *DROP* (for dropping), *RJCT* (for rejecting), *ALLW* (for allowing) or *TLLW* (for two ways allowing). Lines expressing this kind of rules are the most long and complex but they are the very heart of a configuration, so the fact that they are more difficult than the others is not really an unexpected thing;

b) **Policy rules**: they begin with keywords *PDRP* (policy drop) or *PRJC* (policy reject) and they are used to create a policy rule in the firewall[4];

c) **Custom rules**: they are introduced by the keyword *CSTM* and they are used to express a custom rule directly written in the chosen target language (see 2.1.1).

This categorization is just the first superficial description of how rules are expressed in the intermediate language. In table 1, each category of rules is exploded in order to provide detailed information about the meaning of each rule element.

Notice that in the column "parameters", the elements are sorted from up to down, meaning that the first row represents the first parameter of a rule and so on. In the formal grammar presented in 3.2.1 it is possible to observe that keywords are separated from the parameters by a colon (:), while parameters are separated from each other by a semicolon (;).

It is now essential to explain how an endpoint looks like; its structure can be deducted by looking at the formal grammar, but since it is the most relevant element in rules, it is appropriate to give a more detailed description about its meaning and not only about its grammatical structure.

Each endpoint is made of three elements, each of them separated from each other by a semicolon (;):

- **Host**: it is the host or network address (but also an interface can be allowed) that is supposed to send or receive the packets that are filtred by the rule;
- **Interface**: it is the interface packets should come from (or going to) to match the rule. This part of the endpoint is peculiar of a Mignis+ configuration;
- **Port**: it is the port specified in the IP protocol header.

Table 2 gives more details about the legal values for each field of an endpoint structure.

4 A policy rule is a rule that is matched when all the other regular rules are not matched

Table 1: Complete rule list of the intermediate language, with parameters and meaning

Category	Keywords	Parameters	Description
Options	OPTN	Option_Name (String)	The option name
		Option_Value (String)	The option value
Interface bindings	BIND	Interface_Name (String)	The physical interface name (e.g., eth0)
		Network_Address (String)	The network IP address that must be bound to the interface in the standard form (e.g., 192.168.0.0/24)
Firewall rules	DROP	Source (Endpoint)	The endpoint the packets are coming from in the rule
		sNAT (Endpoint)	The source NAT that must be applied in form of an endpoint
		Destination (Endpoint)	The endpoint the packets are going to in the rule
		dNAT (Endpoint)	The destionation NAT that must be appliad in form of an endpoint
		Protocol	The protocol that must be used in order to match the rule
		Additional rule (String)	An additional part of the rule that must be added in the translation process, written directly in the chosen target language
	RJCT	Same parameters and descriptions as DROP	
	ALLW	Same parameters and descriptions as DROP	
	TLLW	Same parameters and descriptions as DROP	
Policy rules	PDRP	Source (Endpoint)	Same as the corresponding parameter in the DROP rule
		Destination (Endpoint)	
		Protocol	
	PRJC	Same parameters and description as PDRP	
Custom rules	CSTM	Custom_Rule (String)	The custom rule written directly in the chosen target language

Table 2: Detailed description of an endpoint

Field	Value	Meaning
Host	String	When the host field is a generic string, then the specified host is an interface. This is possible when the user written Mignis rule filters packets that simply come from (or are sent to) a specific interface regardless of any other information. This is possible only in Mignis because in Mignis+ the interface is specified in a dedicated field
	IP Address	A host IP address or a network IP address that are supposed to be the packet source (or destination). When a host IP is used, then it is preceded by the string h– while for network IP addresses it is used the n– prefix
	ANY	The keyword ANY appears in this field when the rules matches for all packets from/to a source/destination
	LOCAL	LOCAL keyword appears in this field when the source or the destination of the packets filtered by the rule is the local machine (i.e. the device that is being used as firewall)
	MASQUERADE	The MASQUERADE keyword appears only when the associated endpoint has been used in the sNAT part of the rule and only to enable the masquerade feature. In presence of this, all the other fields of the endpoint must be empty strings
	ϵ[5]	The empty string appears in this field only when the whole endpoint is empty. This happens when no NAT has been specified in the rule
Interface	String	A string in this field represents the physical interface name (e.g., eth0) where the filtered packets must come from or be sent to. This field is populated with a string only for Mignis+ configuration file

5 ϵ denotes the empty string

Field	Value	Meaning
Interface	ϵ	The empty string in this field means that the input file was a Mignis configuration file, so the localization of the policy is not active since it is an exclusive Mignis+ feature;
Port	Integer	The Port field is always populated by an integer value which is the port specified in the IP protocol header. The dummy value 0 means that no port has been specified in the rule

It is easy to notice that a rule expressed with the intermediate language always keeps track of all the possible information. For example if no sNAT has been specified, the corresponding rule parameter in the intermediate representation will be an empty endpoint (which means empty strings in the host end interface fields and a 0 in the port field).

Skipping parameters is never allowed

Skipping a parameter in the intermediate language because it is empty is never allowed (as shown in the formal grammar in 3.2.1).

As an example, the Mignis configuration file shown in listing 2.1 is compiled into the intermediate representation shown in listing 3.1.

It is easy to notice that, as already stated, all the keywords are separated from the parameters list in the statements by a colon (:) and each parameter is separated from each other by a semicolon (;) and that no parameter is omitted even if it carries no actual information (so that all the statements belonging to the same category have the same length).

Listing 3.1: The intermediate representation of Mignis input shown in listing 2.1

```
1  OPTN:default_rules;no
2  OPTN:logging;no
3  OPTN:established;yes
4  BIND:eth2;0.0.0.0/0
5  BIND:eth0;10.0.0.0/8
6  BIND:eth1;172.22.0.0/16
7  DROP:eth1;;0;;;0;eth2;;0;;;0;ANY;
8  DROP:ANY;;0;;;0;n-192.168.1.0/24;;0;;;0;ANY;
9  ALLW:ANY;;0;;;0;h-10.0.0.3;;80;h-1.2.3.4;;80;ANY;
10 ALLW:eth1;;0;;;0;h-10.0.0.2;;8080;;;0;TCP;
```

```
11 ALLW:eth0;;0;MASQUERADE;;0;eth2;;0;;;0;ANY;
12 PRJC:ANY;;0;eth0;;0;ANY
13 CSTM:-A INPUT -p tcp --dport 7792 -j LOG --log-prefix "PORT 7792 "
14 CSTM:-A INPUT -p tcp --dport 7792 -j ACCEPT
```

3.2.3 *More in depth: explaining the language's design*

So far we have introduced the formal grammar of the Mignis(+) intermediate language and we have discussed the meaning of the various instruction/keywords together with their parameters. However, nothing have been said about a more theoretical but very important topic: the reasons why it has been designed this way.

Creating a new language which is nothing but an equivalent (and, in this case, lower level) representation of another language is a tricky task since great attention must be paid to many little things that could lead to problems if badly designed. That is why we now present the reasons that led us to design the intermediate language this way and not in another form.

Structure of a whole configuration

Since this language is used to provide a lower level representation of a Mignis(+) configuration file easier to translate into a final target language, it has been thought to have a very regular and predictable structure.

It is not hard to spot that a complete firewall configuration written in the intermediate language is essentially a list of instructions: each row represents something that ranges from an option activation to a custom rule specification, passing through regular firewall rules.

Each row begins with a keyword and ends with a newline (\n) so that the translation process described in chapter 5 can be performed by simply iterating on each row of the configuration and considering one row at a time.

This approach has been chosen because it is both efficient and convenient: it is efficient because the intermediate representation must be scanned by the translator only once and it is convenient because each row produces its effect on the final result at the very moment it is picked up by the translator and when

this latter passes to the next row, there is nothing left to do with the current one. In other words, each row is completely independent from the rest of the whole configuration.

Structure of each row

As we said, each row is independent from all the others: to achieve something like this, each row must be complete in the sense that everything that must be expressed by that row is to be put in its structure without expecting anything to be done by the other rows.

Each row begins with a keyword which is always made by a four letter word. The choice of making all keywords of the same length has been taken in order to create a language which is extremely predictable; since the keyword is the element that expresses what the row is meant to represent within the whole configuration, this is the first element that the translator has to analyze for each row. It is enough to extract the substring made of the first four characters of the current row, to get the keyword and this can be done without any exception.

As seen in 3.2.1 and 3.2.2 each row has a variable number of parameters (depending of what category the keyword belongs to). These parameter can be immediately extracted from the row by taking everything that is put between a colon (:) and a newline character (\n).

It can be said that the parameter list of a row is separated from the keyword by a colon and it continues until the whole row ends with the newline terminator.

The use of a separator is not really needed

We chose this row structure because it is a very regular one and it is extremely easy to get the keyword and the parameter list. The use of a separator between the keyword and the parameter list is not really needed since the length of each keyword is constant and known but it has been put anyway in order to have a more readable language even if it has not been created to be used by human beings.

The separator is a colon character because it is a character that should never appear in the rest of the row.

Structure of the parameter list

The number of parameters depends on the keyword that appears in a row. An *OPTN* row has fewer parameters than an *ALLW* row, and more precisely[6]:

- ***OPTN* keyword**: two fields;
- ***BIND* keyword**: two fields;
- **Keywords belonging to the *Firewall rules* category**: fourteen fields;
- **Keywords belonging to the *Policy rules* category**: seven fields;
- ***CSTM* keyword**: one field.

Fields are all separated from each other by a semicolon (;). It is thus very easy to extract each field from the parameter list since almost all programming languages provides library function to split a string on a separator.

The sorting of parameters within a row is the one shown in table 1 and when an Endpoint must be specified, the internal sorting of this latter element is the one shown in table 2.

These choices have been made because this way it is easy to extract all the fields and since the sorting is fixed and known it is very easy to get complete parameters when they are split in more than one field. In addition to this, the shown sorting has been chose because it is very natural: for firewall rules, for example, the first four parameters are endpoints with this sequence: source, source NAT, destination and destination NAT; such sequence is rather predictable and this is the reason why it has been chosen.

Finally, there is another thing to take into account: the number of parameters (and thus fields) in a row is fixed and, as already stated, depends on the keyword appearing on that row. This always applies regardless of whether a certain element has been written in the original Mignis(+) configuration or not. When an element is omitted in Mignis(+) it means that the rule does not depend on that element; for

6 In this context the number of parameters is actually the overall number of fields. So, for instance, when an Endpoint parameter appears, fields are actually three (see 3.2.2) and here they counts as three parameters

instance if no port is written in the source, it means that the rule matches for all ports.

In the intermediate representation no field is omitted

When an element is omitted in Mignis(+), a special value is assigned to the field that is in charge of represent that element in the intermediate representation; for example, when no port is written in Mignis(+), a dummy value corresponding to 0 is put in the port field of the associated endpoint parameter.

This is a very important choice because it enhances the orderliness of the intermediate representation and when rules are translated we always know where a certain field is located, regardless of the presence of the other fields. In other words, we always know, for instance, that the host of the destination endpoint of an *allow* rule is located in the seventh position among all the fields in the row that represents that rule. This allows to write a better code in the translator (see 5).

Conclusion about the intermediate language design

What we have just presented allows to assert that the design of the intermediate language discussed so far has been the result of a deep analysis and all the choices have been made with a specific goal.

While it is true that this is not the only possible intermediate language and other choices could have been made, it is also true that with our choices we achieved a very predictable language that allows for an easy translation to be performed by the frontend component.

To sum up, the fact that rows are all independent from each other enhances the overall efficiency, while the fact that each row has a fixed structure with fixed length keywords allows for a very easy information retrieving at each iteration.

Eventually, the easy structure of the parameter list and the fact that, given a keyword, the number of field is fixed, allows for an extremely easy process of parameter evaluation when translating a row into a rule written in the final target language.

Because of all these reasons, we can affirm that this intermediate language, with the described grammar and rule meaning, is absolutely fit to represent a Mignis(+) configuration file at a lower level and that it allows to make the final translation process much easier.

3.3 PRIORITY AND OVERLAPPING ISSUES

Mignis and Mignis+ are quite different with respect to all the other firewall configuration languages: rules are normally written with a specific order and swapping two rules could completely change the behaviour of the firewall, while a Mignis(+) configuration is made of a set of rules in which the order is not relevant[1].

In the classical firewall configuration languages, rules are evaluated with a *first come first served* approach; the consequences of such behaviour is that when at least one rule overlaps with another we always can predict what the firewall will do and it corresponds to what is written in the first rule among the overlapping ones that are issued to the firewall.

With Mignis(+) rules order must not affect the final behaviour of the firewall

But since Mignis(+) works with "unsorted" set of rules, this can create some problems because we cannot tollerate that a rule written before another, which is overlapping, could "hide" the latter.

In order to have a better understanding of the problem, consider this simple example: suppose there is a host (h_1) with IP address 10.0.0.5 which obviously belongs to network n_1 identified with net IP address 10.0.0.0/24; suppose also that there is another host (h_2) with IP 192.168.1.10 that belongs to the network n_2 identified with net IP address 192.168.1.0/24.

Suppose now that we want to forbid the forwarding of packets from network n_1 to network n_2 but, at the same time, we want to allow h_1 to connect to h_2.

With a classic firewall configuration language, it is enough to put the *allow* rule before the *forbid* rule: when a packet specifies in its header that the source is h_1 and the destination is h_2 the *allow* rule is matched and since it is written before the *forbid* rule, everything works fine as expected.

When it comes to Mignis(+), however, this is (at least in theoretical terms) no longer possible: the network administrator must be allowed to write rules in whatever order without affecting the final behaviour. However, this is impossible: Mignis(+) must be translated into a target language and this latter is affected by the order of the rules.

We can avoid this overlapping problem by giving a priority to *forbid* rules; in Mignis(+) these rules are the *DROP* (and *REJECT*) rules and they are actually given a higher priority with respect to the *ALLOW* rules[1].

The intermediate language is at a lower level with respect to Mignis(+) and so it does not take into account *set* of rules but it is closer to the real firewall configuration language and it is a list of rules in which the order is relevant; in order to be equivalent to a Mignis(+) configuration, in an intermediate representation the set of *DROP* rules must be put before the se oft *ALLOW* rules so that the priority of the former is kept higher than the latter.

DROP rules must be put before ALLOW rules

Consider now another example (graphically shown in figure 2): a firewall must be set up to forward packets coming from the outside world to two hosts that are located in the private network, willing to accept connections from the same port. The external hosts do not know the internal addresses of the hosts they are allowed to connect to, and so they use the public IP address. This is clearly a situation in which a destination NAT must be set up.

But there is a problem: the public IP address is unique and since the ports to be used are the same for both internal hosts, there is a conflict because the external address should be translated into both internal addresses.

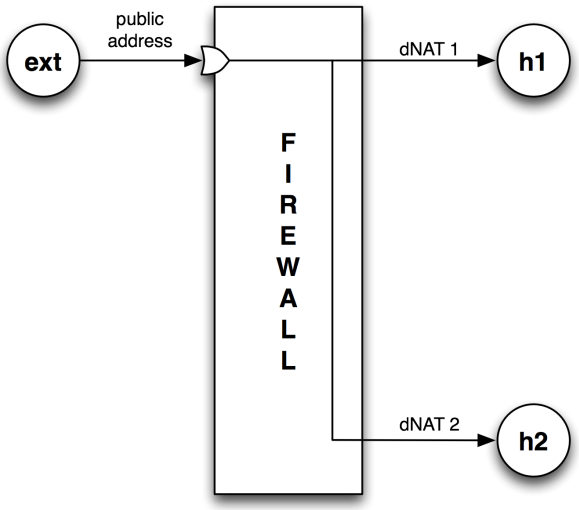

Figure 2: Graphical representation of two conflicting dNAT rules

This is just an example of how overlapping rules can lead to bad problems and this is the reason why they should be avoided. When found they must be blocked or, at least, reported.

Such situations are troublesome when dealing with all firewalls configuration language, but they are particularly annoying when dealing with Mignis(+) that deals with set of rules and not list of rules.

There are many cases, other than the shown examples, in which an overlap can be caught and these include: when two rules share the same source and destination, when two rules share the same source and destination but one of them does not require a destination NAT, when two rules share the same source and one of them is valid for any destination and other similar cases.

It is important to stress that priority and overlaps are not things directly related to the intermediate language but since this latter is at a lower level, it is important, in order to keep the intermediate representation equivalent to the original Mignis(+) one, that the compiler (described in 4.3.2) catches the overlaps and report that there is something that probably has been badly configured. In addition to this, the compiler must give a higher priority to *DROP* and *REJECT* rules and put them before all the *ALLOW* rules, regardless of the actual positioning in the input file.

When rules do not overlap, their ordering is not relevant anymore if not for efficiency reasons. For instance if a host is allowed to connect to a network, it does not matter whether the corresponding rule comes first or last in the allowing rule list (as long as there is no overlap).

However, if this rule is written as the last one and it is very often matched, it would be probably appropriate to consider putting it at the first place in order to enhance efficiency.

4

LEXICAL ANALYSIS, PARSING AND COMPILATION TOWARDS THE INTERMEDIATE LANGUAGE

Mignis(+) is not a programming language but it is a language used to define firewall rules. However, no actual firewall uses Mignis(+) natively so in order to use Mignis(+) in the real world, it must be first translated into a target language; Mignis(+) has its own grammar and a specific lexicon that must be used to correctly transform a Mignis(+) configuration file in an actually working firewall configuration. The translation to the target language is not made directly but it passes through the so-called *intermediate representation*.

Even the easiest and most intuitive language has its own grammar and it is essential to know it in order to write things in the correct way. Even more important, when designing a compiler, is the deep analysis of the grammar as an indispensable part of the project since that grammar must be implemented using existing tools.

Implementing a wrong or imprecise grammar means to obtain completely unpredictable results, which is definitely not what we want to achieve.

In this chapter the lexicon and the formal grammar of the Mignis language is presented and discussed together with the description of the design and creation of the whole backend component, from the lexer to the actual compiler.

4.1 LEXICAL ANALYSIS

By looking at listing 2.1, it is easy to spot that a Mignis configuration file is made essentially by a big number of identifiers and these latter can be used to give a name to an interface or introduce an alias associated to an IP address (see 4.2.1 for the discussion of the formal grammar). Obviously, there are some operators and some keywords as well.

In the lexical analysis phase all these elements must be correctly recognized in order to transform a generic sequence of characters into a sequence of tokens that represents a Mignis configuration file using its lexicon.

4.1.1 *Lexemes, tokens and patterns*

Categories of lexemes

A Mignis(+) configuration file, from a lexical point of view, is made by six categories of lexemes:

1. **KEYWORDS**: this is a set of ten strings that are used to begin a section of the configuration file, select a protocol or refer to the local machine;
2. **IDENTIFIERS**: they are strings used to give a name to the interfaces, define aliases and set options;
3. **IPS**: IPs are network and host addresses used rather frequently in a configuration file; this category includes also the lexemes related to ports;
4. **OPERATORS**: they are characters or sequences of characters used to write the rules;
5. **CUSTOM AND ADDITIONAL RULES**: They are strings that must be concatenated in the final output of the compiler as complete extra rules (directly written in the target language) or as a part of a specific rule;
6. **COMMENTS**: just like any other programming or configuration language.

Category number 1,2,4 and 6 are very similiar to the ones that can be found in any ohter programming language. Indeed, an identifier in Mignis(+) is not different from an identifier in, for instance, C or Java. Same concept applies to the operators category.

Comments are used and behave exactly the same, compared to any other programming or configuration language and are actually stripped off by the lexical analyzer.

Categories 3 and 5 are, on the other hand, much more related to the Mignis(+) language.

IPs are, in fact, the most common value that can be associated to an identifier (see 4.2.1) and they are written with a peculiar (but standard) form.

They are of two different types: host IP and network IP: the former refers to the IP adresses that are used to identify a single machine within a network, while the latter refers to IP addresses that are used to identify and define a whole network.

Custom and additional rules are used to add a complete rule at the end of a configuration or to add elements to a specific rule, respectively. They are needed because there could be some cases in which something more than what Mignis(+) directly provides must be explicitly stated in the rules.

Custom rules are written directly in the target language

This is also the most debatable category of lexemes in the whole Mignis(+) language because when it appears in a configuration file, then the final target language is fixed.

Indeed these "extra rules" are written directly in a target language and this does not allow for any choice other than the used target language. The backend component, however, will not report any error since it is in charge of lexing and parsing only a Mignis(+) configuration and nothing is known at this point about the final target language. It is assumed, when custom or additional rules are used, that the user writes these elements after having already chosen the final target language and that the compiler will be coherently configured to produce, as the final output, the same target language.

On the other hand, it can be very useful to write custom rules.

Tokens

The lexemes belonging to the categories previously described must be recognized in order to return the associated tokens. In table 3 there is a complete list of all the tokens belonging to the Mignis(+) language.

Table 3: List of all the tokens in the Mignis(+) language

Lexeme category	**Token**	**Description**
KEYWORDS	OPTION	Token associated with the lexeme "*OPTIONS*", used in the configuration file to begin the section dedicated to the options declaration
	INTERFACE	Token associated with the lexeme "*INTERFACES*", used in the configuration file to begin the section dedicated to the interfaces declaration
	ALIAS	Token associated with the lexeme "*ALIASES*", used in the configuration file to begin the section dedicated to the aliases declaration
	FIREWALL	Token associated with the lexeme "*FIREWALL*", used in the configuration file to begin the section dedicated to the firewall rules declaration
	POLICY	Token associated with the lexeme "*POLICIES*", used in the configuration file to begin the section dedicated to the firewall policies declaration
	CUSTOM	Token associated with the lexeme "*CUSTOM*", used in the configuration file to begin the section dedicated to the custom rules declaration
	LOCAL	Token associated with the lexeme "*local*", used when it is required to refer to the local machine
	TCP	Token associated with the lexeme "*tcp*", used when rules applys only when the TCP protocol is used
	UDP	Token associated with the lexeme "*udp*", used when rules applys only when the UDP protocol is used
	ICMP	Token associated with the lexeme "*icmp*", used when rules applys only when the ICMP protocol is used

Lexeme category	Token	Description
IDENTIFIERS	IDENTIFIER of *string*	Token associated with a string that must begin with a lower or upper case; the token is always together with the string recognized as an IDENTIFIER
IPS	HOST_IP of *string*	Token associated with a string representing a regular ip made of four numbers separated by a dot (.) character; the token is always together with the string recognized as a HOST_IP
	NET_IP of *string*	Token associated with a string representing a network address made of four numbers separated by a dot (.) character followed by a slash (/) character and another number; the token is always together with the string recognized as a NET_IP
	PORT of *int*	Token associated with a colon (:) followed by an integer value representing the port; the token is always together with the integer (without colon characters) recognized as a PORT
OPERATORS[1]	ALLOW	Token associated with the lexeme ">", used to express that packets are allowed to pass
	TWALLOW	Token associated with the lexeme "<>", used to express that packets are allowed to pass in both direction
	DROP	Token associated with the lexeme "/", used to express that packets not allowed to pass and they are dropped[2]
	REJECT	Token associated with the lexeme "//", used to express that packets are not allowed to pass and they are rejected[3]

1 For the details concerning the meaning and use of operators, see 2.1.1
2 Drop means that the packet cannot pass and no response is sent to the source host
3 Reject means that the packet cannot pass but an ICMP "destination-unreachable" message is sent to the source host

Lexeme category	Token	Description
OPERATORS	STAR	Token associated with the lexeme "∗", used as a wildcard meaning "any host or network"
	AT	Token associated with the lexeme "@", used exclusively in Mignis+ (see 2.1.2 for greater details)
	LBRACK - RBRACK	Tokens associated to the lexemes "*[*" and "*]*", respectively, used when a source or destination NAT is required
	DOT	Token associated to the lexeme ".", used in a source NAT context to express that a masquerade is required
CUSTOM ADDITIONAL RULES	CUSTOMRULE of *string*	In the custom section of a configuration file, it is possible to insert raw rules written directly in the target language and these are associated with this token (which always comes together with the string recognized as a CUSTOMRULE)
	FORMULA of *string*	When a rule already written using Mignis(+) syntax needs more details written in the raw target language, these details are associated with this token (which always comes together with the string recognized as a FORMULA)

Patterns and regular expressions

As already stated in 2.2.1, all the described lexemes can be recognized as istances of the tokens presented in table 3 by using Deterministic Finite Automata.

It is then rather obvious that patterns that formally describe lexemes are written using regular expressions.

These regular expressions are basically of two kinds:

- **Kind 1**: this simpler kind is the one used for the keywords and for the operators.

Basically these are fixed and trivial strings that must be matched exactly "as is".
The following is the complete list of strings that must be matched (on the left side of the arrow) in order to return the associated token (on the right side):

$$
\begin{aligned}
\text{OPTIONS} &\Longrightarrow \text{OPTION}\\
\text{INTERFACES} &\Longrightarrow \text{INTERFACE}\\
\text{ALIASES} &\Longrightarrow \text{ALIAS}\\
\text{FIREWALL} &\Longrightarrow \text{FIREWALL}\\
\text{POLICIES} &\Longrightarrow \text{POLICY}\\
\text{CUSTOM} &\Longrightarrow \text{CUSTOM}\\
\texttt{local} &\Longrightarrow \text{LOCAL}\\
* &\Longrightarrow \text{STAR}\\
@ &\Longrightarrow \text{AT}\\
[&\Longrightarrow \text{LBRACK}\\
] &\Longrightarrow \text{RBRACK}\\
. &\Longrightarrow \text{DOT}\\
> &\Longrightarrow \text{ALLOW}\\
<> &\Longrightarrow \text{TWALLOW}\\
/ &\Longrightarrow \text{DROP}\\
// &\Longrightarrow \text{REJECT}\\
\texttt{tcp} &\Longrightarrow \text{TCP}\\
\texttt{udp} &\Longrightarrow \text{UDP}\\
\texttt{icmp} &\Longrightarrow \text{ICMP}
\end{aligned}
$$

- **Kind 2**: this kind is the one used to describe lexemes that are not fixed strings (e.g., IP addresses, identifiers). These elements are always different but they have a common underlying formal representation.
  The following is the complete list of regular expressions (on the left side of the arrow) that return, when matched, the associated token (on the right side):

$$
\begin{aligned}
[A-Za-z][A-Za-z0-9\_]^* &\Longrightarrow \text{IDENTIFIER} && (4.1)\\
\mid [\hat{}\ \backslash n]^* &\Longrightarrow \text{FORMULA} && (4.2)\\
[A-Za-z0-9 \text{:space:} - \ ''\ ]^+ &\Longrightarrow \text{CUSTOMRULE} && (4.3)\\
([0-9]\{1,3\}\ .)\{3\}\ [0-9]\{1,3\} &\Longrightarrow \text{HOST\_IP} && (4.4)\\
([0-9]\{1,3\}\ .)\{3\}\ [0-9]\{1,3\}\ /\ [0-9]\{1,2\} &\Longrightarrow \text{NET\_IP} && (4.5)\\
:\ [\text{:space:}]^*\ [0-9]^+ &\Longrightarrow \text{PORT} && (4.6)
\end{aligned}
$$

where:

- $[c_1 - c_2]$ means any character between c_1 and c_2;
- $[\hat{} \, c_1]$ means any character other than c_1;
- $e_1\{n, m\}$ means that e_1 must be repeated at least n and at most m times while $e_2\{i\}$ means that e_2 must be repeated exactly i times;
- e_1^* means zero or more (without limits) repetitions of e_1, while e_2^+ means one or more (without limits) repetitions of e_2;
- [:space:] is the usual white space;
- \n is the usual newline character.

All these patterns must be implemented in a programming language in order to actually make them recognize the lexemes and return the correct associated patterns.

4.1.2 *Implementation with OCamllex*

All the details about the lexical analyzer presented so far are exclusively theoretical and we need now to actually implement something that given a character stream as its input, returns a sequence of tokens as its output.

This can be done by using *OCamllex*, a lexer tool contained in the OCaml distribution and described in 2.2.2.

In order to create the Mignis(+) lexical analyzer, the first thing to do is the implementation of the regular expressions described in 4.1.1. In listing 4.1 all the regular expressions are defined using the OCamllex syntax described previously.

Listing 4.1: Definition of the regular expressions describing the non-trivial Mignis(+) lexemes

```
1  (* Auxiliary expressions *)
2  (* Single number *)
3  let number = ['0' - '9']
4
5  (* Ingeger *)
6  let int = number+
7
8  (* Octet in IPs *)
9  let octet = number number? number?
10
11 (* Subnet mask *)
```

```
12 let subnet = number number?
13
14 (* Main definitions *)
15 (* A generic identifier, used to give names to interfaces, hosts and so on *)
16 let identifier = (['A' - 'Z'] | ['a' - 'z'])
17                 (['A' - 'Z'] | ['a' - 'z'] | '_' | int)*
18
19 (* A custom rule, with spaces and symbols *)
20 let custom = (['A' - 'Z'] | ['a' - 'z'] | int |
21               ' ' | '-' | '\"')+
22
23 (* An additional rule for mignis+ *)
24 let additional = '|' [^'\n']*
25
26 (* A full network address *)
27 let net_addr = octet '.' octet '.' octet '.' octet '/' subnet
28
29 (* A full host address *)
30 let host_addr = octet '.' octet '.' octet '.' octet
31
32 (* Port recognition pattern *)
33 let port = ':' (' ')* int
```

Lines 1-13 defines auxiliary regular expressions, used in the main definitions (lines 14-31), in order to make everything much more readable.

Some of this expressions are worthy spending a few words:

- At line 9 an octet is defined; this particular expression matches a sequence of at least one and at most three numbers (defined at line 3). This will be used in the definition of the IP address in which there are four octets separated by a dot (.) (lines 27 and 30);
- At line 12 a subnet is defined; this is made by at least one and at most two numbers. This will be used in the definition of the network IP address in which a normal host IP is followed by a slash (/) and a subnet (line 27);
- At line 16 an identifier is defined as a sequence of alphabetical characters, integer numbers (defined at line 6) and underscores. An identifier must begin with an alphabetical character;
- At line 20 a custom rule is defined as a sequence of alphabetical characters, integer numbers, spaces, minuses and quotas. The length is at least one;
- At line 24 an additional rule (token FORMULA) is defined as a pipe (|) followed by a sequence of characters and symbols with the exception of the new line;

- At line 33 a port is matched as a colon (:) followed by an integer number. Between the colon and the numbers there could be some white spaces.

When a *port* or an *additional* expression is matched, the bound value includes the separation character and the leading white spaces. For example the string ": 25" matches against the *port* expression but what we want to bind to the PORT token is just the number 25, not the colon or the white spaces. This is why the function shown in listing 4.2 has been put in the OCamllex header section.

Listing 4.2: Function used to clean up the matched strings from unwanted characters and white spaces together with the current state type definition

```
1 (* This function is used to get a string without spaces and any instances *)
2 (* of character c at the string beginning *)
3 let rec clean s c =
4   let len = String.length s in
5   if s.[0] == c || s.[0] == ' ' then clean (String.sub s 1 (len - 1)) c
6   else s
7
8 (* This is the type used to keep trace of the current state *)
9 type state_t = MainConf | CustomRules
```

Identifiers and custom rules overlap almost perfectly from a lexical point of view

A much more treacherous point is the one concerning the *identifier* versus the *custom* expressions recognition: they partially overlap. The difference between the two is just the fact that the *custom* expression allows for spaces and some extra characters, while the *identifier* one allows only for underscores other than alphanumeric characters. As long as a string contains only alphanumeric characters, it can be matched by both expressions. In this specific case the attribution of a specific priority isn't enough to solve the problem. From a lexical point of view, the two elements actually sometimes perfectly overlap and the only way to distinguish them is to check in which part of a Mignis(+) configuration file we are. This can be done only by exploiting the formal grammar of a Mignis(+) configuration file (discussed in 4.2.1)

In a Mignis(+) configuration file there cannot be any identifier in the section dedicated to the custom rules, so the trick to solve the overlap problem is to keep track whether we are in the "CUSTOM" section or not. If we are in the "CUSTOM" section, then we always try to recognize the `custom` expression; we recognize the `identifier` expression in all the other sections.

To do so, we first define a new type called `state_t` that allows only for two values (`MainConf` or `CustomRules`, see line 9 in listing 4.2). After that, we initialize a variable, in the trailing section, of type `state_t` to the value `MainConf`. Whenever a "CUSTOM" keyword is found, before returning the CUSTOM token, the variable value is switched to `CustomRules` allowing to call the `cstrule` entry point in the future. Whenever a "OPTIONS" keyword is found when the current state is `CustomRules`, the variable is switched again to `MainConf`, so that we will be able to enter again the `main` entry point.

In listing 4.3 it is shown the variable initialization and the function `next_token`, that uses the value of the variable to decide which function must be called.

The complete lexical analyzer source code is shown in listing A.1.

Listing 4.3: Lexical analyzer trailing section

```
1  (* Current state is MainConf *)
2  let current = ref MainConf;;
3
4  (* To get the next token, the current state is matched and decisions are *)
5  (* taken *)
6  let next_token =
7    (fun lexbuf -> match !current with
8     | MainConf -> main current lexbuf
9     | CustomRules -> cstrule current lexbuf)
10 ;;
```

4.2 PARSING

The lexical analysis discussed so far is just the first step in a compiling process and thus it is only the first activity done by the backend component of the new Mignis(+) compiler.

If the lexical analysis does not fail, its output (the stream of tokens) is passed directly to the parser in order to perform the activity known as parsing (briefly discussed in 2.3).

Parsing takes as its input the stream of tokens and returns an Abstract Syntax Tree.

We now introduce the formal grammar and the structure of the AST, before discussing the actual implementation of the parser.

4.2.1 *Formal grammar*

As described in 2.3.1, a context-free grammar has to be used to correctly define and describe a language structure. Truth be told, Mignis(+) is not a programming language and it has not a lot of balanced items. However, since the tools used to actually implement the parser (see 4.2.3) are designed to work with context-free grammars, the Mignis(+) language will be here described using such kind of grammars.

In order to formally define a context-free grammar, we must define the four sets that together form the 4-tuple $\mathfrak{G}$.

It is easy to see that the alphabet Σ_M is basically the set of tokens returned by the lexical analyzer described in 4.1.1 and more specifically in table 3. So we can formally say that:

$$\begin{aligned}\Sigma_M = \{&\text{OPTION, INTERFACE, ALIAS, FIREWALL, POLICY, CUSTOM,}\\ &\text{LOCAL, TCP, UDP, ICMP, IDENTIFIER, HOST\_IP, NET\_IP, PORT,}\\ &\text{ALLOW, TWALLOW, DROP, REJECT, STAR, AT, LBRACK, RBRACK,}\\ &\text{DOT, CUSTOMRULE, FORMULA}\}\end{aligned}$$

This is not surprising since the parser has to do its job on the stream of tokens returned by the lexical analysis, so all the terminal symbols in set Σ_M must belong to the set of tokens representing the lexicon of Mignis(+)[4]

The set $\mathcal{N}_M$ for the non-terminal symbols has been defined as follows in order to write the production rules in a much more readable way.

$$\begin{aligned}\mathcal{N}_M = \{&\texttt{config}, \texttt{Options}, \texttt{OptionList}, \texttt{InterfaceDecl}, \texttt{InterfList},\\ &\texttt{AliasDecl}, \texttt{AliasList}, \texttt{FirewallConf}, \texttt{FirewList}, \texttt{Endp}, \texttt{Nt}, \texttt{Prtc},\\ &\texttt{Formula}, \texttt{If}, \texttt{Prt}, \texttt{PolicyConf}, \texttt{PolicyList}, \texttt{CustomRules},\\ &\texttt{CustomList}\}\end{aligned}$$

4 There is one more token (EOF) that has not been written here because it is not a token belonging to the language but a technical token used in the actual implementation

Once the terminal symbols Σ_M set and the non-terminal symbols $\mathcal{N}_M$ set have been defined, it is possible to define the set of production rules $\rightarrow_M$.

For the sake of readability the production rules are simply listed and not written in form of set. In uppercase are written the terminal symbol belonging to Σ_M while in lowercase (with capitalized first letter of each word) are written the non-terminal symbol belonging to $\mathcal{N}_M$.

The use of a star character (*) after a symbol (or set of symbols) is to be interpreted as "zero or more repetitions" as if it were a regular expression. This little abuse of notation improves the overall readability.

$$
\begin{aligned}
\mathit{config} \rightarrow & (\mathit{Options}\ \mathit{InterfaceDecl}\ \mathit{AliasDecl}\ \mathit{FirewallConf} \\
& \quad \mathit{PolicyConf}\ \mathit{CustomRules})^* && (4.7)\\
\mathit{Options} \rightarrow & \ \mathrm{OPTION}\ \mathit{OptionList} && (4.8)\\
\mathit{OptionList} \rightarrow & (\mathrm{IDENTIFIER}\ \mathrm{IDENTIFIER})^* && (4.9)\\
\mathit{InterfaceDecl} \rightarrow & \ \mathrm{INTERFACE}\ \mathit{InterfList} && (4.10)\\
\mathit{InterList} \rightarrow & (\mathrm{IDENTIFIER}\ \mathrm{IDENTIFIER}\ \mathrm{NET\_IP})^* && (4.11)\\
\mathit{AliasDecl} \rightarrow & \ \mathrm{ALIAS}\ \mathit{AliasList} && (4.12)\\
\mathit{AliasList} \rightarrow & (\mathrm{IDENTIFIER}\ [\mathrm{HOST\_IP} \mid \mathrm{NET\_IP}])^* && (4.13)\\
\mathit{FirewallConf} \rightarrow & \ \mathrm{FIREWALL}\ \mathit{FirewList} && (4.14)\\
\mathit{FirewList} \rightarrow & (\mathit{Endp}\ \mathit{Nt} \\
& \quad [\mathit{ALLOW} \mid \mathit{TWALLOW} \mid \mathrm{DROP} \mid \mathrm{REJECT}] \\
& \quad \mathit{Nt}\ \mathit{Endp}\ \mathit{Prtc}\ \mathit{Formula})^* && (4.15)\\
\mathit{Endp} \rightarrow & \ [\mathrm{IDENTIFIER} \mid \mathrm{HOST\_IP} \mid \mathrm{NET\_IP}]\ \mathit{If}\ \mathit{Prt} \\
& \mid \mathrm{LOCAL}\ \mathit{Prt} \\
& \mid \mathrm{STAR} && (4.16)\\
\mathit{Prt} \rightarrow & \ \mathrm{PORT} \mid \epsilon && (4.17)\\
\mathit{If} \rightarrow & \ \mathrm{AT}\ \mathrm{IDENTIFIER} \mid \epsilon && (4.18)\\
\mathit{Nt} \rightarrow & \ (\mathrm{LBRACK}\ [\mathrm{DOT} \mid \mathit{Endp}]\ \mathrm{RBRACK}) \mid \epsilon && (4.19)\\
\mathit{Prtc} \rightarrow & \ \mathrm{TCP} \mid \mathrm{UDP} \mid \mathrm{ICMP} \mid \epsilon && (4.20)\\
\mathit{Formula} \rightarrow & \ \mathrm{FORMULA} \mid \epsilon && (4.21)\\
\mathit{PolicyConf} \rightarrow & \ \mathrm{POLICY}\ \mathit{PolicyList} && (4.22)\\
\mathit{PolicyList} \rightarrow & (\mathit{Endp}\ [\mathrm{DROP} \mid \mathrm{REJECT}]\ \mathit{Endp}\ \mathit{Prtc})^* && (4.23)\\
\mathit{CustomRules} \rightarrow & \ \mathrm{CUSTOM}\ \mathit{CustomList} && (4.24)\\
\mathit{CustomList} \rightarrow & \ \mathrm{CUSTOMRULE}^* && (4.25)
\end{aligned}
$$

Another little abuse of notation is the the use of square brackets to indicate the choice among a set of possibilities (like rules 4.13 or 4.15). ϵ means the usual empty string; when a * is used, the case in which there are zero repetitions of a symbol or a set of symbols implies that the accepted string is ϵ.

Finally, the starting simbol $\mathcal{S}_M$ is the non-terminal `config` which represents a whole configuration file.

The Mignis(+) formal grammar $\mathfrak{G}_M$ is then formally defined as:

$$\mathfrak{G}_M = (\mathcal{N}_M, \Sigma_N, \rightarrow_M, \mathcal{S}_M)$$

It is easy to see, by simply taking a more careful look at listings 2.1 and 2.2, that $\mathfrak{G}_M$ perfectly defines the Mignis(+) syntax; however, there is a point that is worthy a few more words of explanation.

Rule 4.19 defines how a NAT is specified in the rules and it allows for an empty string (meaning that there is no NAT) or for a more complex structure when a NAT is actually requested.

More specifically, the rule wants a DOT or an `Endp` to be specified between the couple LBRACK – RBRACK; while there is no problem when a DOT is specified, some troubles could rise when `Endp` is matched: rule 4.16 defines `Endp` and it allows also for the LOCAL and STAR terminal symbols to be used and this is obviously something we do not want to allow when defining a NAT.

NAT rule is less strict than it should be

The reason of this is that it is convenient to use the `Endp` rule to specify a NAT because a legal NAT is nothing but an endpoint with some limitations (and thus less rules have to be written, simplifying the overall complexity of the parser). The unwanted combinations must be eliminated when actually compiling the Mignis(+) source in the intermediate language (see 4.3) and not catched by the parser.

4.2.2 *The "Abstract Syntax Tree"*

When the parser succeeds to parse a whole file configuration, it must return an AST that represents the whole configuration in a form of a tree data structure.

Given that we are not dealing with a complete programming language, the structure of the tree is rather simplified.

The complete strucure, defined with the code shown in listing A.3, can be summarized as follows:

- **Firewall list**: each firewall configuration found in the input file is represented by an element of this list. Each entry is made of:
  - **Option list**: each element written in the options section of a firewall configuration is here represented by a 2-tuple made of 2 strings;
  - **Interface list**: each element written in the interface declaration section of a firewall configuration is here represented by a 3-tuple made of 3 strings;
  - **Alias list**: each element written in the alias declaration section of a firewall configuration is here represented by two possible structures:
    - **Hostalias**: a 2-tuple made of 2 strings (the second string describes a host IP address);
    - **Netalias**: as the Hostalias but here the second strings describes a network IP address;
  - **Rule list**: this is the very heart of a firewall configuration since it describes the list of rules that must be applied. Each rule is an entry of this list. There are 4 possible structures but they are identical, except for the name of the structure itself. The 4 structure names are: *Allow*, *Twallow*, *Drop* and *Reject*. The complete structure is made of a 6-tuple made of (in the exact order):
    - **Endpoint**: it represents the source endpoint and to describe all the possible kinds of endpoints it can be one of these structures:
      - **Name**: a 3-tuple made of 1 strings, 1 sub structure and an integer (name, interface and port); the substructure can be made of:
        - **If**: it represents an interface and it is always made of a string for the interface's name;
        - **Noif**: no interfaces;

        - **Ip**: a 3-tuple made of 1 strings, 1 sub structure (as the ones in previous point) and an integer (ip, interface and port);
        - **Local**: it is used to represent the *local* keyword and it is made of an integer to specify the port;
        - **Star**: used to represent the star wildcard and it is always alone;
    * **Nat**: it represents the sNAT and it can be made of one of these following structures:
        - **Nat**: it appears when a Nat has been declarated and it is made of a complete endpoint (defined as the Endpoint structure explained above);
        - **Masquerade**: it appears (alone) when a masquerade is requested;
        - **Nonat**: it appears when no NAT is required;
    * **Nat**: the same as the Nat explained above except for the fact that this is used for the dNAT;
    * **Endpoint**: the same as the Endpoint explained above but used for the destination;
    * **Protocol**: it represents the protocol to be matched by the rule. It can be *Tcp*, *Udp*, *Icmp* or *Noprotocol*;
    * **Formula**: It represents the optional additional rule and it can be one of the following:
        - **Formula**: it represents the additional rules and it is made of a string (which is the additional rules itself);
        - **Noformula**: no additional rules;
- **Policy list**: a list in which each element is called *Default*, made of a 4-tuple with this structure:
    * **A Policy operator**: it represents the operation requested by the policy and it can be *Pdrop* or *Preject*;
    * **Endpoint**: it represents the source in the general policy and it is an Endpoint like the ones explained above;
    * **Endpoint**: the destination in the general policy;

    * **Protocol**: the protocol to be matched. It is described above in greater details;

  – **Custom list**: a list of custom rules (made of strings).

It is easy to notice that the complete structure of the AST is rather complex even if it is simplified with respect to the one of a complete programming language. The small configuration shown in listing 4.4 can be represented with the abstract syntax tree shown in figure 3.

Listing 4.4: A simple Mignis configuration to be represented as an AST

```
1  OPTIONS
2  logging no
3
4  INTERFACES
5  lan     eth0    10.0.0.0/24
6  wan     eth1    0.0.0.0/0
7
8  ALIASES
9
10 FIREWALL
11 lan [.] >       wan
12
13 POLICIES
14 *       /       *
15
16 CUSTOM
```

As shown in figure 3, a small Mignis configuration generates a rather huge AST, but this structure takes with it all the information needed by the compiler to produce an intermediate representation of a Mignis(+) configuration.

The AST is complex but it provides all is needed

Notice that, for instance, no port has been specified in the rules, but in the *Endpoint* structure there is a "dummy" 0 (which indeed is a special value that means "no port"). The same applies to other parts of the configuration: for example the absense of a protocol specification is highlighted by the presence of a *Noprotocol* structure.

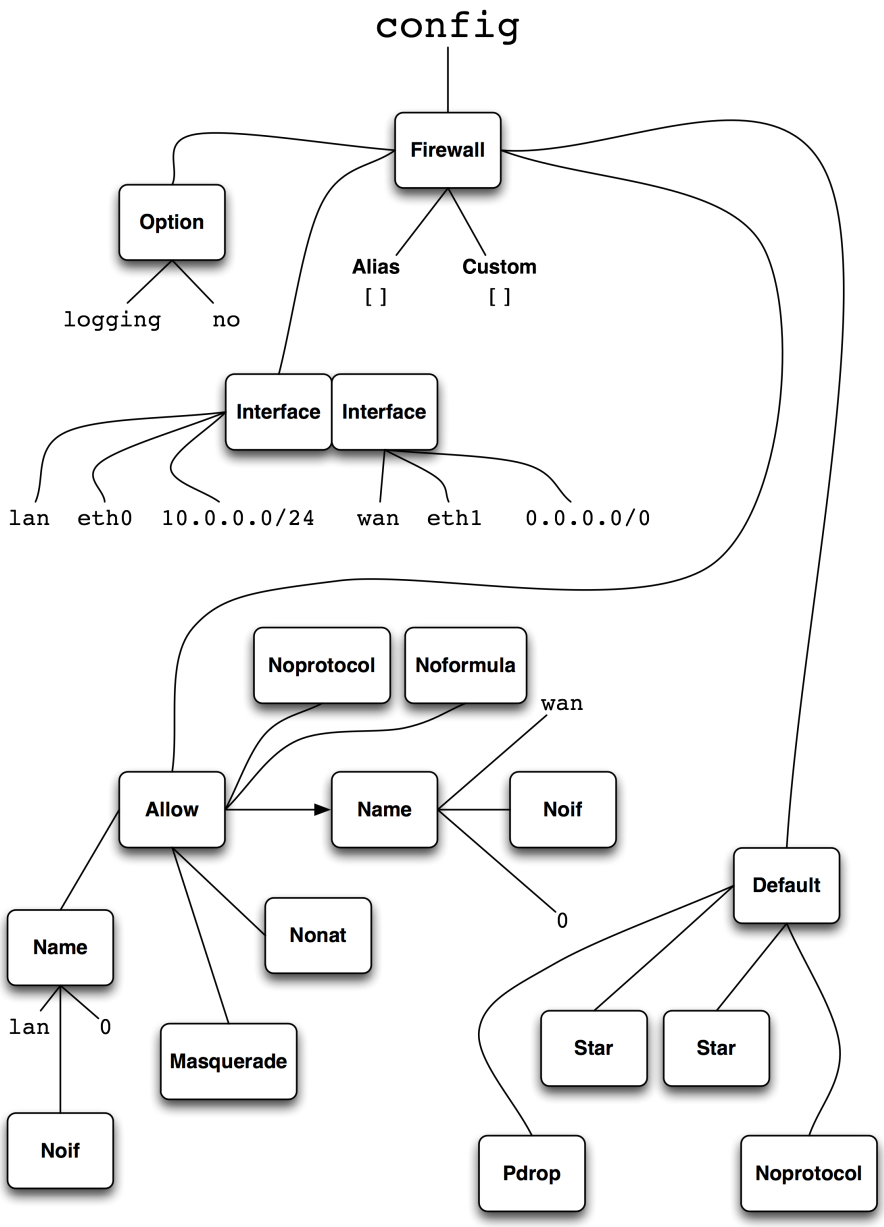

Figure 3: Representation of the Abstract Syntax Tree generated by the parser for code in listing 4.4

4.2.3 *Creating the parser with OCamlyacc*

The parser described so far must be able to recognize the context-free grammar described in 4.2.1 and it has to return an Abstract Syntax Tree compliant with the structure discussed in 4.2.2.

It is now time to describe how to use OCamlyacc to actually implement all the thing, seen so far only from a theoretical and formal point of view.

To create the Mignis(+) parser, the context-free grammar discussed in 4.2.1 must be implemented using the structure of an OCamlyacc definition (described in 2.3.2).

Each non-terminal belonging to the set $\mathcal{N}_M$ will be defined in the rules section of the definition and each of the defined rule will actually implement, with the OCamlyacc syntax, a the production rule belonging to the set $\rightarrow_M$. An example of some rules definition is shown in listing 4.5.

Listing 4.5: Small snippet of the rules section in the parser definition

```
1  /* The interface part begins with the keyword INTERFACE */
2  InterfaceDecl:
3    INTERFACE InterfList                { $2 }
4  ;
5  /* This is the structure of an interface declaration */
6  InterfList:
7    IDENTIFIER IDENTIFIER NET_IP  InterfList
8                                       { Interface($1, $2, $3)::$4 }
9   |                                   { [] } /* Empty list */
10 ;
```

In addition to this, all the terminal symbols belonging to set Σ_M are here defined at the beginning of the Ocamlyacc configuration, in the declarations section; this is a crucial part of the entire compiler since these terminal symbols are also the tokens returned by the lexical analyzer (see 4.1.1). The code of this essential section is shown in listing 4.6.

It is also possible to notice at line 19 that the non-terminal `config` has been chosen as the start symbol and it is associated with the type *Mast.config* (line 20) which is the name for the root node of the Abstract Syntax Tree described in 4.2.2.

The complete source code for the parser is shown in listing A.2.

Listing 4.6: Terminal symbol and token definition in the parser source code

```
/* Main key words */
%token OPTION INTERFACE ALIAS FIREWALL POLICY CUSTOM LOCAL STAR AT
/* Ids, addresses and ports */
%token <string> FORMULA
%token <string> CUSTOMRULE
%token <string> IDENTIFIER
%token <string> NET_IP
%token <string> HOST_IP
%token <int> PORT
/* Operators */
%token ALLOW DROP REJECT TWALLOW
/* Brackets and dots */
%token LBRACK DOT RBRACK
/* Protocols */
%token TCP UDP ICMP
/* EOF */
%token EOF

%start config
%type <Mast.config> config
```

4.3 SCOPING AND FINAL COMPILING

Once the AST has been generated by using the parser described and discussed in details in 4.2, the next step of the compiling process is the one concerning the scoping and the compiling itself.

All the activities described so far are actually preparatory to the final step that is going to be discussed in this section.

After having successfully performed the lexical analysis and the parsing, we are sure that what has been found within a configuration file complies with the formal structure of Mignis(+). The filling of the scoping tables are a further step which is very useful to create a complete ambient that allows to compile without needing to continuously scan the AST.

However, the formal correctness of a configuration file does not imply that a configuration is a *correct* configuration. This is only an apparent contradiction: the formal correctness is about complying to a formal lexicon and grammar, while writing a correct configuration means to write something that is coherent and, eventually, makes sense.

The AST must now be scanned in order to create a complete ambient (scoping tables) that must be used in the final step: the production of the firewall configuration written in the intermediate language discussed in chapter 3. This latter step can be done only after having checked that the configuration is correct from a semantic point of view.

4.3.1 *Scoping*

Since Mignis(+) is not a programming language, using the word "scoping" is an abuse of naming; however, there are interfaces and aliases that can be declared and then they must be resolved when actually used in firewall rules. For this reason is very convenient to scan the whole AST in order to create scoping tables to be used by the final compiler when in need of a name resolution.

In the source code file for this component there is a rather big number of definitions of functions that are used to extract the information from the AST. All these information are then stored in variables that serve as scoping table. In listing 4.7 the declaration of these variables is shown.

Listing 4.7: Declaration of the variable used as scope tables

```
1  (* Options *)
2  let option_list:((Mast.id * Mast.id) list) list ref = ref [];;
3  (* Interfaces *)
4  let interface_list:((Mast.id * Mast.id * Mast.net_ip) list) list ref = ref [];;
5  (* Aliases *)
6  let alias_list:((Mast.id * string) list) list ref = ref [];;
7  (* Firewall rules *)
8  let rule_list:(Mast.op list) list ref = ref [];;
9  (* Policies *)
10 let policy_list:(Mast.policy list) list ref = ref [];;
11 (* Custom rules *)
12 let crule_list:(Mast.crule list) list ref = ref [];;
```

The first thing to notice is that each of these variables is declared as a list of lists. The reason is very easy: a Mignis(+) file can consists of several firewall configurations, so the most logical thing to do when populating each scoping table is to:

- Use an outer list in which each element refers to a firewall configuration. So, for instance, the element in the list with index 3, refers to the 4th firewall configuration;

- Use another inner list to keep track of all the elements in a specific section of a firewall configurations. So each element of the outer list is a list as well.

For the sake of clarity, here there is a rather easy example: suppose that in the *ALIASES* section of the 2nd firewall configuration the 3rd alias is associated to the string *my_computer*; the identifier *my_computer* is to be found in the *alias_list* list using index 1 for the outer list (2nd firewall) and the index 2 for the inner list (3rd declared alias).

Each inner list is made of n-tuples, depending on what they have to represent:

- *option_list*: each element of the inner list is made of a 2-tuple of identifiers, namely strings;
- *interface_list*: each element of the inner list is made of a 3-tuple, 2 identifiers (string) and 1 network IP address (still a string but with a predefined structure);
- *alias_list*: each element of the inner list is made of a 2-tuple, 1 identifer (string) and 1 generic string (without restrictions). This latter type has been used because the alias can be both a network or a host IP address;
- *rule_list*: each element of the inner list is made of just 1 rule taken directly from the AST (see 4.2.2);
- *policy_list*: like *rule_list* but with the structure taken from the policies branch of the AST;
- *crule_list*: like the two previous elements above, but with the structure taken from the custom rules branch of the AST.

The function shown in listing 4.8 is used to perform the lexical analysis and the parsing of a file whose name is passed as an argument. This function is essential because it is the ones that taken a file as an input, returns the AST that must be scanned.

Listing 4.8: Function lex_and_parse

```
let lex_and_parse file =
  try config next_token (read_source file) with
  | Failure error -> failwith ("Lexer error: " ^ error)
  | Parsing.Parse_error -> failwith ("Parse error")
;;
```

The function shown in listing 4.9 is the main function for the creation of the ambient made of the scoping tables. It essentially generates an AST from an input source file and then initializes all the scoping tables to the empty list. Once the scoping tables are all ready, the function *create_conf_table* is called.

Listing 4.9: Main function for the scoping table population process

```
let start source =
  let ast = lex_and_parse source in
  option_list := [];
  interface_list := [];
  alias_list := [];
  rule_list := [];
  policy_list := [];
  crule_list := [];
  create_conf_table ast
;;
```

The source code of function *create_conf_table* is shown in listing 4.10. A big number of auxiliary functions are then called in order to populate all the scoping tables and making them ready to be used by the actual compiler in the next and final step.

Listing 4.10: Function create_conf_table

```
let rec create_conf_table (ast:Mast.config) =
  match ast with
  | fw::rest                          -> conf_firewall fw;
                                         create_conf_table rest

  | []                                -> ()
;;
```

The complete source code for this part of the compiler is shown in listing A.4.

4.3.2 *Final compiling process*

Now that the scoping tables are filled and ready to be used, we must perform the last step of this compilation of a Mignis(+) configuration file towards the intermediate language.

But before actually producing the final intermediate representation of the configuration file given as input, the compiler must perform some semantic verifications. In particular it must check: whether there are overlaps between the rules or not, if Mignis rules appear together with Mignis+ rules, if NATs are mistakenly configured and so on.

Once all these tests are passed, the set of strings written in the intermediate language can be actually produced as output and written into one or more files.

In listing 4.11 are shown the declarations of variables that are used by the final compiler.

Listing 4.11: Variable declarations for the final component of the compiler

```
1  (* List of strings in which we save the compiled configuration for each *)
2  (* firewall *)
3  let compiled:(string list) ref = ref [];;
4
5  (* List of operands to keep track of overlapping rules *)
6  let overlaps:(string list) ref = ref [];;
7
8  (* Are we on a Mignis or Mignis+ configuration file? *)
9  type conf_t = Mignis | MignisPlus | NotSet;;
10 let conf:(conf_t) ref = ref NotSet;;
```

The variable *compiled* is the most important one, because it keeps track of all the produced configurations: when a configuration has been successfully compiled, all the strings that are part of that configuration are merged together in a single string and added to this list in which each entry is the complete configuration of a firewall.

The variable *overlaps* is another list of strings and it is, at some extent, redundant with *compiled*: also this variable keeps track of the rules of the current configuration but each entry is a single rule. Whenever a new rule must be compiled, it is checked that there isn't any overlap between all the rules accepted up to that moment. All the accepted rules are put in this variable.

The variable *conf*, associated with the special type defined at line 9, is used only to keep track whether we are compiling a Mignis or a Mignis+ file. At the beginning the value must be NotSet, but when the first rule is found the variable is set to Mignis or MignisPlus. In listing 4.12 is shown how this task is performed.

Listing 4.12: Function set_interface

```
1  (* This function is the one that is allowed to track if we are on a Mignis *)
```

```
2  (* or Mignis+ configuration file and it keeps things coherent *)
3  let set_interface inf amb nat =
4    match inf with
5    | Mast.Noif                    ->
6      if !conf = MignisPlus && nat = false then
7        failwith("Mignis and Mignis+ rules cannot be used together")
8      else if !conf = NotSet then
9        begin
10         conf := Mignis;
11         ""
12       end
13     else
14       ""
15   | Mast.If(id)                  ->
16     let resolved = find id amb in
17     if resolved = "" then
18       failwith("Interface not declared")
19     else
20       if !conf = Mignis && nat = false then
21         failwith("Mignis and Mignis+ rules cannot be used together")
22       else if !conf = NotSet then
23         begin
24           conf := MignisPlus;
25           resolved
26         end
27       else
28         resolved
29 ;;
```

A Mignis+ configuration is recognized when rules are localized, in other words when endpoints are written with the interface we are expecting packets to come from or to go to. If this information is not present (`Noif` structure in the AST, see 4.2.2), then we are in a Mignis configuration.

This approach is implemented in the *set_interface* function as shown in listing 4.12: the match structure at line 4 recognizes whether an interface has been specified or not; when no interface has been written in the current rule, the variable *conf* is checked and if its value shows that we are compiling a Mignis+ configuration file, an exception is raised (lines 6-7). The same thing happens when an interface has been written but we are compiling a Mignis configuration file (lines 20-21). This enforcement is not applied when an endpoint is put within a NAT declaration.

A configuration file must use only Mignis or Mignis+, mixing the two is forbidden

When the compiler starts, the variable *conf* is set to the `NotSet` value and the first rule decides whether we are in a Mignis or a Mignis+ configuration file: lines 8-12 choose for Mignis, while lines 22-26 choose for Mignis+.

It is important to forbid mixed rules because Mignis+ has some restrictions compared to regular Mignis (see 2.1.2).

Another peculiar restriction of Mignis+, is that rules must be all positive; this means that we can't use the drop or reject operator. This is done by the *create_rules* function (a short, but significant, snippet is shown in listing 4.13): when a Drop or Reject is found in the list of rules previously parsed, there is a check whether we are on a Mignis or Mignis+ configuration. If the latter is verified, then an exception is raised (lines 6-8, for a Drop operator, the Reject is not different).

Listing 4.13: Small snippet of the create_rules function

```
1  let rec create_rules rls ambient =
2    warning := "";
3    match rls with
4    ...
5    | Mast.Drop(from,snat,dnat,dest,prt,frm)::rest
6                                    -> if !conf = MignisPlus then
7                                         failwith("Mignis+ does not allow " ^
8                                                  "for negative rules");
9                                       let operands =
10                                        set_op from snat dnat dest prt frm ambient
11                                                                          in
12                                      if check_overlaps (Str.split comma operands)
13                                        !overlaps = true then
14                                        begin
15                                          overlaps := !overlaps @ [operands];
16                                          if !warning <> "" then
17                                            Printf.printf "Warning: %s" !warning;
18                                         "DROP:" ^
19                                         operands ^
20                                         "\n" ^ (create_rules rest ambient)
21                                       end
22                                     else
23                                       failwith(!warning);
24   ...
25   | []                              -> ""
26 ;;
```

An interesting thing done by the function *create_rules* for all operators, is to check if the rules overlap and it also prints a warning message when something wrong has been found (regardless of the fact that we are compiling a Mignis or Mignis+ configuration file).

When a rule is not overlapping another rule, but it is wrong from a conceptual point of view (for example when a destination NAT is declared as a masquerade), then not only a message is printed but the whole compiling process is made to fail. This can be seen at lines 12-23.

It is easy to see that every rule is checked by the *check_overlaps* function (line 12), but this function is not the one that actually performs all the tests. The function that actually checks all the rules is shown in listing 4.14. This latter function was

originally thought only for overlaps detection, but then it has been enhanced to perform tests on a more conceptual level.

Listing 4.14: Function chk_rule

```
let chk_rule r1 r2 =
  let s1 = List.nth r1 0 ^ "," ^ List.nth r1 1 ^ "," ^ List.nth r1 2 in
  let s2 = List.nth r2 0 ^ "," ^ List.nth r2 1 ^ "," ^ List.nth r2 2 in
  let sn1 = List.nth r1 3 ^ "," ^ List.nth r1 4 ^ "," ^ List.nth r1 5 in
  let sn2 = List.nth r2 3 ^ "," ^ List.nth r2 4 ^ "," ^ List.nth r2 5 in
  let d1 = List.nth r1 6 ^ "," ^ List.nth r1 7 ^ "," ^ List.nth r1 8 in
  let d2 = List.nth r2 6 ^ "," ^ List.nth r2 7 ^ "," ^ List.nth r2 8 in
  let dn1 = List.nth r1 9 ^ "," ^ List.nth r1 10 ^ "," ^ List.nth r1 11 in
  let dn2 = List.nth r2 9 ^ "," ^ List.nth r2 10 ^ "," ^ List.nth r2 11 in
  if s1 = s2 && d1 = d2 &&
     sn1 = sn2 && sn1 = ",,0" &&
     dn1 = dn2 && dn1 = ",,0" then
    begin
      warning := !warning ^
        "A rule with same source and destination has been found\n";
      true
    end
  else if s1 = s2 && d1 = d2 &&
          dn1 <> dn2 &&
          (dn1 = ",,0" || dn2 = ",,0") &&
          sn1 = sn2 then
    begin
      warning := !warning ^ "A rule with dNAT is overlapping a generic rule\n";
      true
    end
  else if s1 = s2 && d1 <> d2 &&
          dn1 = dn2 && dn1 <> ",,0" then
    begin
      warning := !warning ^ "A rule with an overlapping dNAT has been found\n";
      true
    end
  else if s1 = s2 &&
          d1 <> d2 &&
          (d1 = "ANY,,0" || d2 = "ANY,,0") then
    begin
      warning := !warning ^ "A rule is overlapping a * destination\n";
      true
    end
  else if sn1 <> ",,0" && dn1 <> ",,0" then
    begin
      warning := "A rule cannot specify a sNAT and a dNAT at the same time";
      false
    end
  else if dn1 = "MASQUERADE,,0" then
    begin
      warning := "Destination masquarade is not allowed";
      false
    end
  else if sn1 = "ANY,,0" || dn1 = "ANY,,0" then
    begin
      warning := "Wildcard * cannot be used in NAT declarations";
      false
    end
  else if (contains sn1 "LOCAL") || (contains dn1 "LOCAL") then
    begin
      warning := "Keyword local cannot be used in NAT declarations";
```

```
57        false
58      end
59    else
60      true
61  ;;
```

At lines 1-9, some OCaml expressions, used later in the various tests, are initialized and at lines 10-61 all the tests are actually perfomed; for example at lines 32-38 rules with the same source endpoint but different destination endpoints in which one of them is the wild card star (*) are detected: a warning is set but the compiling process is not set to fail. At lines 49-53 NAT declarations with a wildcard $*$ are detected: with such an error the compiling process is set to fail.

Drop and Reject rules have priority over the other rules

The overlap detection is very important because a Mignis(+) configuration is made by a *set* of rules and not by a *list* of rules[1] as seen in 3.3. Overlapping rules are a problem: the first that matches usually wins on all the others that overlap. Another important thing is that in the compiling process the rules that drop or reject packets have a higher priority over the other rules (i.e. rules that allow packets to flow); so even if a drop rule is written as the last rule, it will be assigned a higher priority (see 3.3).

The compiler performs alias resolution

Mignis(+) language allows for the definition of aliases and names for the interfaces, but in the intermediate representation is not possible to keep these symbolic names. This is the reason why the compiler performs the symbolic names resolution, in other words aliases are resolved to the host or network IP they are related to, and interfaces' names are resolved to the physical name of the Network Interface Card they are bound to. Interfaces' names are related not only to the physical name of the NIC but also to a network IP address that is assigned to the same NIC. That is why in the intermediate language there are instruction that create this bound (see 3.2.2).

Listing 4.15: Function find

```
1  (* Aux function used to find an alias or an interface *)
2  (* Please note that aliases have a higher priority *)
3  let rec find needle ambient =
4    let alias = List.nth !Scope.alias_list ambient in
5    let result_alias = find_alias alias needle in
6    if result_alias = "" then
7      let interface = List.nth !Scope.interface_list ambient in
8      find_interface interface needle
9    else
```

```
10 ||     result_alias
11 || ;;
```

In listing 4.15 the code of the function *find* is shown: this function resolves an alias to an IP (host or network) or an interface's name to the physical name of the NIC. There are two auxiliary function that are called and they actually search in the scoping table the alias or the interface's name (within the current firewall configuration: the scope of an alias is the configuration it is declared in). Notice that an interface's name could collide with an alias: in this case the alias has the higher priority and hides the interface's name.

When rules passes all the non-fatal tests, then the configuration is actually compiled into the intermediate language discussed in chapter 3. The complete source code to perform this operation is shown in listing A.5.

The full compiler (lexer, parser and compiler together with the components used to fill the scoping tables) is used by a main program that allows the user for an interaction with it. The source code of this small program is shown in A.6.

5

THE FINAL TRANSLATOR

So far we have introduced the Mignis(+) intermediate language and we have discussed how to compile a Mignis(+) configuration file into an intermediate representation.

We have seen how the compiling process performs a lexical analysis and a parsing together with a semantic analysis in order to produce an intermediate representation that is equivalent to the original Mignis(+) configuration.

Nothing or very little has been said about the frontend component, namely the final translator that is capable of producing the final configuration files written in the chosen target language.

The final translator is an essential part of the new Mignis(+) compiler since it takes an intermediate representation as input and it transforms it in the final firewall configuration that can be used on an actual device.

In this chapter we describe the design of the final translator and we provide a completely working example of translation into a Netfilter configuration using iptables commands.

5.1 STRUCTURE OF THE FRONTEND COMPONENT

The goal of the frontend component is to take an intermediate representation and transform it into a final firewall configuration written in an actual target language.

We assume that the input is correct and produced by the backend component and never written by hand. As already stated both in 3.1 and 3.2.1, the intermediate language is not intended to be written or read by the human being and the backend component's duty is to check if the Mignis(+) configuration is correctly written by the user and to produce the intermediate representation when no errors have been found. When this intermediate representation is produced by the compiler

described in chapter 4 we can assume that there is no mistakes and the frontend component can start and perform its translation into the target language.

We can thus affirm two things on the frontend component:

1. The intermediate language is never lexed or parsed by the translator;
2. In order to fulfill its task, the translator simply needs to iterate on the intermediate representation rows (see 3.2.3) and translate them into rules written in the target language.

Point 1 directly derives from the fact that there is the assumption that the intermediate representation is always produced by the backend component; when this is not the case, the translator could produce unpredictable results.

Point 2 derives from the structure of the intermediate language and from point 1: we already discussed the fact that each row in the intermediate representation is independent from all the other (see 3.2.3) and since no lexing or parsing is performed, all that is needed is just taking each row and translate it into the final target language independently from all the other rows. This can be done by simply looping row by row.

OCaml is not really a widespread programming language as it is ranked below the 50th place in the index of the most popular programming languages (October 2016). If the frontend component were written using this language, then there are few chances that independent programmers could expand this project by adding a new target language which is not yet supported.

On the other hand, Python is the fifth most popular programming language (October 2016): writing the final translator using this language allows for a higher probability that a network administrator in need of a new target language, independently expands the frontend component[1].

In addition to this, the Python programming language is particularly easy to learn and use since it has a simple syntax and many built-in data structure (such as lists or dictionaries)[9]. This feature makes it even more suitable for the developing of this part of the project.

1 The popularity indexes of the programming languages are taken from http://www.tiobe.com/tiobe-index/ on October 7th 2016

These are the reasons that led us to use Python to implement the frontend component instead of continuing to use OCaml.

In this section we introduce the structure of the frontend component, describing which approach has been used and giving the details about its fundamental elements.

5.1.1 *Object-Oriented approach*

This component is essentially a program that taken an input written in a fixed language (the intermediate language, see chapter 3), returns an output written in another language. Its main feature, however, is that it has been designed to be able to support multiple target languages. This implies that given a fixed input, the output can be different at each run depending on the chosen target language. Obviously, the supported target languages are a finite subset of all the possible firewall configuration languages.

From a design perspective, there are two different kind of tasks that the frontend component is required to perfom:

1. Reading a configuration written in the intermediate language and doing some operations that do not depend on the final target language;
2. Translating what has been read at point 1 into the final target language.

It is easy to seen that these two different tasks can be easily implemented with an object-oriented approach.

Figure 4 shows the very simple classes structure that is needed to implement the frontend component.

Basically there is a class called *Generic Engine* that is in charge of defining all the operations that are to be performed regardless of the final target language (see 5.1.2 for more details).

This class is extended by a number of subclasses that are in charge of defining how the intermediate language has to be actually translated into the final target language.

Eventually, there is a *Main program* that will use a Generic Engine object to perform the translation.

From the Main program point of view, the translator is nothing but a Generic Engine object that knows how to load an intermediate representation and how to translate it into a target language chosen by the user. The Main program is obviously the interface the user interacts with and so the Main program is in charge also to correctly manage the target language choice.

The Generic Engine is an abstract class

It is not difficult to notice that with a structure like this the Generic Engine class is an abstract one, so that it cannot be directly instantiated; this is because of a very simple reason: the Main program uses a Generic Engine, believing that it is the translator and that it knows how to do the job; however, this element only knows how to perform the basic operations and demands to the subclasses the actual implementation of the translation.

So the Main program will actually use a subclass of the Generic Engine so that we are sure that the instance of the object is capable of performing the final translation.

It has been chosen to implement the translator with this object oriented programming approach because this allows for a very simple but powerful thing: in order to support a new target language it is enough to create a new subclass of the Generic Engine abstract class; this new subclass must be capable of translating the intermediate language into the new target language and there is no need to implement again all the basic and shared operations that are needed to prepare the translation job. The Generic Engine has everything that is needed to allow the subclasses to perform a good translation.

This is an extremely easy approach that simply exploits the object-oriented programming concepts, but it allows to expand the new Mignis(+) compiler to support new target languages by simply add another subclass.

This feature is really useful when it comes to add a new target language: there is no need to re-invent the wheel and the only thing the programmer will have to work on is how to translate the intermediate language into the new target language; this makes the task easier and faster.

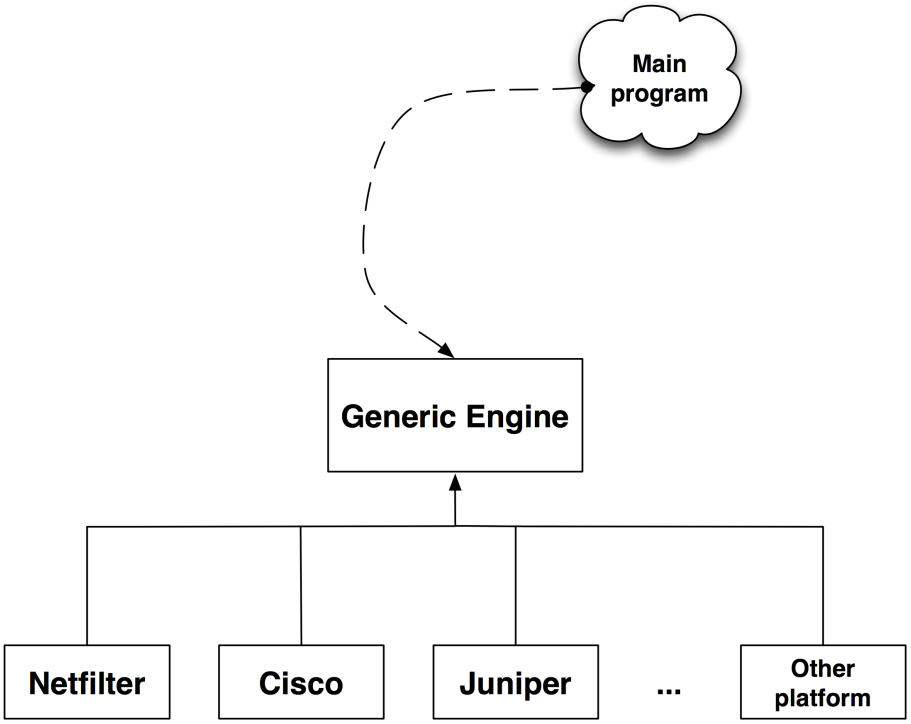

Figure 4: Structure of classes in the translator program

5.1.2 *The "generic engine" abstract class*

As shown in figure 4 and partially explained in 5.1.1, the translation is fully managed by an object that is called "Generic Engine" which is actually an abstract class that is in charge of performing all the things that need to be done regardless of the final target language.

The following are all the methods that are implemented in the `GenericEngine` class:

- **read_files**: this method takes no arguments and it is used to read all the configurations files (written in the intermediate language) located at a predefined directory. It returns a list of all the read configurations in the form of strings;
- **parse_line**: this method takes one argument that is the line that needs to be "parsed". Notice that the line is not actually parsed, the name is a small abuse of naming. This method simply returns a 2-tuple: the first element is a string that is the keyword of the line, the second is a list of strings which is the list of the parameters associated with the line itself (see 3.2.2);
- **get_rule_details**: this method takes one argument that is the parameters list of a row. It returns another list made of tuples in which the parameters are grouped depending on the their meaning. For example the first three parameter of a firewall rule are grouped in the "source" endpoint. It returns the list of grouped parameters;
- **switch_elements**: this method takes two arguments, the first is a string and the second is a list of 2-tuples. The first element of the tuple is a string treated as a pattern, the second element is a string that must take the place of all the occurrences of the pattern. It returns a string which is the original given one with all the patterns specified in the list substituted by the associated string. This method is used when the operator <> appears in the Mignis(+) configuration: when this happens there is the need of producing two rules in the final target languages and what was the source in the first rule, is actually the destination in the second; with this method it is easy to swap things;

- **translate**: this is the abstract method and its goals and features are better described in 5.1.3;
- **compile**: this is the main method of the class and it takes no arguments. It is in charge of doing everything concerning the translation of the intermediate representation into the chosen target language. It returns the number of the translation successfully performed.

The whole source code of the Generic Engine class (GenericEngine) is shown in listing B.1; however, it is particularly useful to comment at least the method compile since it is the very heart of the the GenericEngine class.

Listing 5.1: Method compile in GenericEngine class

```
1  def compile(self):
2      # The complete file name structure is
3      # ../final/fw<index>.<targe_language>
4      prefix = self.directory + "final/fw"
5      suffix = "." + self.language
6
7      # We get all the configurations written in the intermediate
8      # representation
9      conf_list = self.read_files()
10
11     n = 0
12     # For all the configurations found
13     for conf in conf_list:
14         final_conf = self.translate(conf)  # Translate it
15         file_name = prefix + str(n) + suffix  # Create the file name
16         try:
17             # Create a new file and write the final configuration in it
18             out_stream = open(file_name, "w")
19             out_stream.write(final_conf)
20             out_stream.flush()
21             out_stream.close()
22             n += 1
23         except IOError as _:
24             # If something goes wrong, skip the file and continue with
25             # the next
26             print("ERR: Skipping output file %s because of an I/O error"
27                   % file_name)
28             continue
29
30     # Return the number of final configurations written
31     return n
```

It is easy to see in the compile method source code, shown in listing 5.1 at lines 4-5 and 15, that the configuration written in the final target language are put in a predefined folder and there is a strong naming convention: files are put in the "final" subfolder located in the directory where the original Mignis(+) configuration files are to be found.

The naming convention for the final files is the following: the string "fw" followed by a progressive number followed by a dot (.) and the name of the target language. For example, the first firewall configuration produced for Netfilter shall have the name fw0.iptables (being iptables the name of the language used for Netfilter).

It is also easy to see how the main directory and the chosen target language are saved in instance variables called directory and language.

Line 9 is in charge of loading all the configurations written in the intermediate language; these configurations are put in files located in the "compiled" subfolder and their names follow this naming convention: the string "fw" followed by a progressive number followed by the string ".config".

The loop shown at lines 13-28 is the heart of the method: translate is called at line 14 and then the result of the translation process is saved into a file. Method translate is an abstract method, so it can be called here because it is in its scope, but we don't know yet how the actual translation will take place.

The name "compile" for this method is actually an abuse of naming since what happens within it is not a compiling process. This name has been used to distinguish it from the method translate.

5.1.3 *The actual translator: the subclasses of the "generic engine"*

GenericEngine class already defines all that is needed to perform a translation from the intermediate representation into the final target language. As explained in 5.1.1, since the final target languages are more than one, the inheritance principles of the object-oriented programming really fit to perform the actual translation into one of these target languages.

The main idea is that the subclasses of the GenericEngine class simply need to define the abstract method translate that they inherit from the super class.

With this approach the translate method can use all the auxiliary methods defined by GenericEngine in order to deal with the rules structure and all the

other things that are not directly related with the translation process in its strictest meaning.

From the user's point of view, there exists only the `GenericEngine` class for all the target languages. All the subclasses can be seen as an element that knows how to actually translate into a specific target language but it is nothing more than a translator.

The translation takes place by iterating on the rows of an intermediate representation

From the method `translate`'s point of view, performing a translation is nothing more than iterating on all the rows found in an intermediate representation and translate each of them into its equivalent written using the target language the subclass is defining.

One of the most important benefit of this approach is that in order to support a new target language, it is enough to extend the `GenericEngine` class and implement the method `translate` to correctly perform the actual translation of the intermediate language into the new target language. The developer does not need to write code to retrieve the intermediate representation since the configuration will be passed as an argument to the `translate` method itself. This latter method will have access to all the already implemented auxiliary methods provided by the super class. This approach fully exploits the object-oriented principles.

If the target language that is managed by a subclass has some "special needs" (meaning that there are some operation to be performed but that they are related only to that specific target language), that subclass can define methods other than the `translate` one that this latter can use in order to perform these special operations.

In listing B.2 it is shown how a minimal example subclass of `GenericEngine` looks like: there is nothing more that the constructor (used to initialise the properties of a `GenericEngine` instance) and the method `translate`. The complexity of this latter method depends on the complexity of the final target language but its duty does not change: it iterates on all rows of the intermediate representation of a configuration and translate them into their equivalents in the target language.

5.2 THE FINAL TRANSLATOR MAIN PROGRAM

The structure of the frontend component introduced in 5.1 must be used by a main program that allows the final user to specify where the configuration files written in the intermediate representation are located and which is the desired target language.

This part has been written using the Python language as a Linux script that can be executed from the command line.

From the final user's point of view, this script simply asks for two command line arguments:

1. The chosen final target language. To this argument can be passed the special value "list" to obtain the list of all the supported target languages;
2. The directory where the Mignis(+) configuration file is located. The frontend component will automatically compute where the intermediate representations are put and will create the folder in which the configurations written in the chosen target language will be saved.

When a wrong target language is chosen (misspelled or simply not yet supported) a specific error is returned, informing the user that the chosen language is not available.

The main program is also responsible to create the "final" subfolder, where the `compile` method (`GenericEngine` class) will put the result of the translation process and if the folder already exists, then it is deleted and then created again to avoid conflicts.

The most important activity performed by the main program is the assignment of the object on which the method `compile` will be invoked; the correct class to instantiate is determined by the user via the command line argument described above at point 1.

It is important to stress that, in order to support a new target language, it is not enough to implement the new subclass (as described in 5.1.3), but this latter must be associated to a language name and correctly instantiated in the main program.

The main program, in the end, calls the `compile` method on the assigned object and with this operation, the translation actually takes place.

The full source code of this script used as the main program is shown in listing B.4.

5.3 AN ACTUAL EXAMPLE: NETFILTER/IPTABLES TARGET

The current official release of Mignis takes as input a Mignis configuration file and produces a list of iptables commands from the set of user defined rules.

It is then reasonable that the first supported target language of this new multi-target compiler is iptables.

This choice has multiple benefits but the biggest one is that we have a point of comparison: the semantic of Mignis has been proved equivalent to the semantic of Netfilter[1] and so if the new compiler is able to produce the same list of rules starting from the very same input configuration, then it means that the new tool is capable of producing fully working and formally correct actual firewall configuration.

The new compiler, unlike the old one, supports both Mignis and Mignis+ syntax; so, it is able to produce iptables configuration files for both Mignis and Mignis+ input configuration.

However, Mignis+ has been presented recently in mid 2016[2], so the final rule translation concerning Mignis+ is still an experimental support and it must be considered a beta (especially concerning the NAT support).

The order of the rules written with iptables by the new compiler could be different from what is returned by the old compiler; this is not a problem since Mignis(+) uses set of rules that must be valid regardless of the rules order (see 3.3) provided that there is absence of overlaps.

Finally, the new compiler always explicitly state the protocol that must be used in order to match a certain rule and when all protocols are accepted (i.e. the user didn't specify any protocol), it explicitly states, unlike what the old compiler used to do, that all protocols are accepted instead of leaving it unspecified.

From a practical point of view, the translation is performed by the `translate` method of the `NetfilterEngine` class (derived from `GenericEngine`). This method does exactly what it is expected to do: it iterates on each row of the intermediate representation of a configuration and translates it (independently from all the other rows) using the iptables commands syntax.

Options are managed setting flags to on or off and the currently supported options are:

- **default_rules**: if set to on, it adds some extra default rules. This option is supported also by the old compiler and its default value is *on*;
- **logging**: if set to on, it adds some extra rules used to log the packet filtering. This option is supported also by the old compiler and its default value is *on*;
- **established**: if set to on, it adds some extra rules used to always allow established connection. This option has been added to support the Mignis+ restriction on established connections. Its default value is *off*.

The full source code for the `NetfilterEngine` class is shown in listing B.3; however, in listing 5.2 are shown some class variables that defines how the rules in iptables format look like. These variables are used to create all the actual rules starting from the intermediate representation.

Listing 5.2: Class variables that are used for the building of iptables rules

```
BIND_ACCEPT = \
    "-A PREROUTING -i {0} -s {1} -j ACCEPT -m comment --comment \"{2}\"\n"
BIND_ANY_ACCEPT = \
    "-A PREROUTING -i {0} -j ACCEPT -m comment --comment \"{1}\"\n"
BIND_ANY_DROP = \
    "-A PREROUTING -i {0} -s {1} -j DROP -m comment --comment \"{2}\"\n"
MANGLE_NAT = \
    "-A PREROUTING {0} {1} {2} {3} {4} -m state --state NEW -j DROP " + \
      "-m comment --comment \"{5}\"\n"
RULE_FWD = \
    "-A FORWARD {0} {1} {2} {3} {4} {5} -j {6} " + \
      "-m comment --comment \"{7}\"\n"
RULE_IN = \
    "-A INPUT {0} {1} {2} {3} {4} -j {5} -m comment --comment \"{6}\"\n"
RULE_OUT = \
    "-A OUTPUT {0} {1} {2} {3} {4} -j {5} -m comment --comment \"{6}\"\n"
RULE_MASQUERADE = \
    "-A POSTROUTING {0} {1} {2} {3} {4} -j MASQUERADE " + \
      "-m comment --comment \"{5}\"\n"
RULE_SNAT = \
    "-A POSTROUTING {0} {1} {2} {3} {4} -j SNAT --to-source {5} " + \
      "-m comment --comment \"{6}\"\n"
RULE_DNAT = \
```

```
24      "-A PREROUTING {0} {1} {2} {3} {4} -j DNAT --to-destination {5} " + \
25        "-m comment --comment \"{6}\"\n"
```

It is easy to notice that these rules are written with a parametric approach, so that the actual values from the rules can be placed instead of the patterns {n}, by using the `format` standard method provided by Python on string objects.

6

CONCLUSIONS AND FUTURE WORKS

In this thesis we have presented a new multi-target compiler for Mignis and Mignis+. The final goal was to produce a tool that is capable of providing a uniform language for firewall rules specification and we achieved this result using a modular approach.

Basically the presented tool is made by two independent parts (the backend and the frontend components) that used together can produce a final configuration that can be issued to a real-life firewall (as long as its target language is supported by this new tool).

The backend component is in charge of checking the formal correctness of the Mignis(+) configuration file, by checking its lexicon and grammar. It also performs a semantic analysis by giving priority to rules and checking overlaps among rules.

The frontend component is in charge of producing the final configuration written in a chosen target language (that must be supported).

Between the two components there is the heart of all the project: the intermediate language. This language has been developed with the goal of allowing the backend component to completely ignore which is the target language chosen by the user and to produce a lower level representation of a Mignis(+) configuration that can be more efficiently translated into a target language by the frontend component that is not supposed to check any formal correctness.

The intermediate language have been designed to represent a configuration as a set of rows, each of them independent from the others, in which everything is always explicitly stated (even if there is nothing to state). This extremely predictable structure makes the intermediate language very easy to translate into a target language.

The overall structure of the project is shown in figure 5.

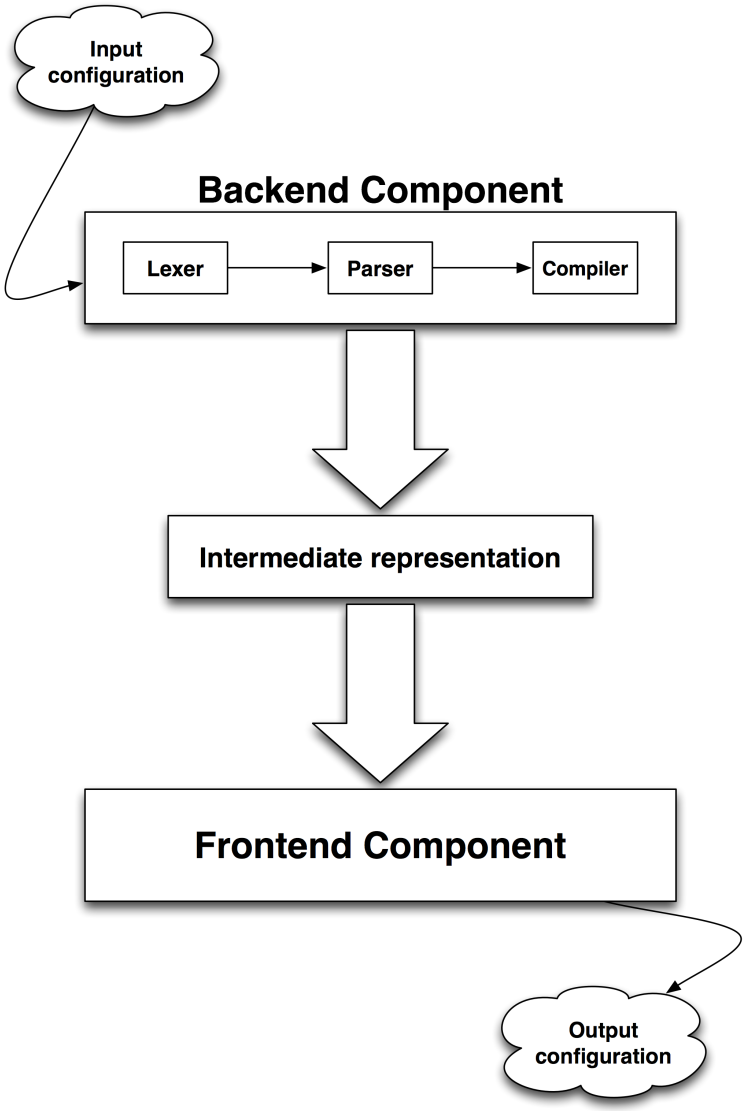

Figure 5: Overall structure of the new multi-target Mignis+ compiler

The fully working compiler, available for the download as an Open Source software at https://github.com/hellslord/mignis-compiler/[1] , is made of these components and of a Python script that acts as a director: it first invoke the backend component and then it calls the frontend components on the former's output so that the latter produces the final configurations. The full source code of this director script is shown in listing C.1.

This software is a new tool (still in development) which is useful for network administrators that need to set up corporate networks with many firewalls potentially based on different platforms, but that would like to use a single language to implement the policies.

The overall number of code lines of this new multi-target compiler is about 1600 (more than the old compiler); however, part of them are not involved in the process of adding another target language because the whole backend component, in charge of checking the formal correctness of a Mignis(+) configuration file and producing the intermediate representation, does not depend on the target language. This means that adding a new target language "costs" only an average number of 450 code lines (which is a great improving of almost one third).

6.1 KNOWN LIMITATIONS

Despite being a fully working tool, this version of a completely new software is still in a development state. It is thus obvious that it has some known limitations and these are:

- At the moment it fully supports only the translation into Netfilter configurations with iptables commands. This is the biggest limitation since the tool has been thought as a multi-target compiler. However, it has been developed so that it is very easy to support new target languages;
- The translation of Mignis+ configurations is, at the moment, still in beta especially for the part concerning NAT rules. This is due to the fact that

1 The version presented in this thesis is the 2.4.1

Mignis+ has been presented very recently and the tool is to be considered experimental;

These limitations does not prevent the use of the new Mignis(+) compiler, but it means that it can be upgraded to work better.

At the moment, however, we are already working to add the support for the Juniper target language.

6.2 FUTURE WORKS

Our goal was to provide a uniform language for firewall configuration by presenting a multi-target compiler. We achieved this result and there is now a fully working software that can be actually used not only in academic contexts but also in real life situations.

However, the work is far from being over: the known limitations described above are to be overcome in the immediate future in order to make this tool actually useful, but there are more theoretical things that need to be addressed and in particular:

1. The semantic of the intermediate language should be further designed and formally described;

2. It should be formally proved that the Mignis(+) semantic is equivalent to the intermediate language semantic, so that the two representation of a configuration can be formally declared as completely equivalent;

3. Overlapping rules are a problem and the backend component inspects the input configuration trying to find them before producing of an intermediate representation. All the trivial overlapping cases are found by the compiler but a deeper analysis, to be done in the immediate future, could help to find out more non-trivial cases that are not currently implemented.

4. An analysis on the efficiency could be useful: the order of the rules is not relevant (when there are no overlaps) but some rules could be matched more often than others. A statistical approach to discover which are the rules to

be pur first in order to reduce the time needed to match them could be very useful.

In particular points 1 and 2 are intended to be addressed in the immediate future in order to formally prove the robustness and correctness of the new tool presented in this thesis.

Appendices

A

COMPLETE SOURCE CODE FOR THE MIGNIS(+) COMPILER (BACKEND COMPONENT)

Listing A.1: File lexer.mll

```
{
  open Parser
  open String

  (* This function is used to get a string without spaces and any instances *)
  (* of character c at the string beginning *)
  let rec clean s c =
    let len = String.length s in
    if s.[0] == c || s.[0] == ' ' then clean (String.sub s 1 (len - 1)) c
    else s

  (* This is the type used to keep trace of the current state *)
  type state_t = MainConf | CustomRules
}

(* Definition of some regular expressions, useful to recognize tokens *)

(* Auxiliary expressions *)
(* Single number *)
let number = ['0' - '9']

(* Ingeger *)
let int = number+

(* Octet in IPs *)
let octet = number number? number?

(* Subnet mask *)
let subnet = number number?

(* Main definitions *)
(* A generic identifier, used to give names to interfaces, hosts and so on *)
let identifier = (['A' - 'Z'] | ['a' - 'z'])
                 (['A' - 'Z'] | ['a' - 'z'] | '_' | int)*

(* A custom rule, with spaces and symbols *)
let custom = (['A' - 'Z'] | ['a' - 'z'] | int |
              ' ' | '-' | '\"')+

(* An additional rule for mignis+ *)
let additional = '|' [^'\n']*

(* A full network address *)
let net_addr = octet '.' octet '.' octet '.' octet '/' subnet

(* A full host address *)
let host_addr = octet '.' octet '.' octet '.' octet

(* Port recognition pattern *)
let port = ':' (' ')* int
```

```
51 |
52 | (* Definition of the lexer rules *)
53 | (* Main entrypoint *)
54 | rule main state_ref = parse
55 |   (* End of file is matched *)
56 |   | eof                          { EOF }
57 |   (* Ignore blank spaces, tabs and CRLF *)
58 |   | (" " | "\t" | "\n")          { main state_ref lexbuf }
59 |   (* Comments *)
60 |   | '#'                          { comment state_ref lexbuf }
61 |   (* Main keywords, the one defining the configuration sections and local *)
62 |   | "OPTIONS"                    { OPTION }
63 |   | "INTERFACES"                 { INTERFACE }
64 |   | "ALIASES"                    { ALIAS }
65 |   | "FIREWALL"                   { FIREWALL }
66 |   | "POLICIES"                   { POLICY }
67 |   (* The custom rules section is a rather special one because custon rules *)
68 |   (* are recognized with a regex that overlaps with the one for the *)
69 |   (* identifiers. So the state is changed in order to persistently use *)
70 |   (* the cstrule entrypoint *)
71 |   | "CUSTOM"                     { state_ref := CustomRules; CUSTOM }
72 |   | "local"                      { LOCAL }
73 |   (* There are a fixed number of recognized protocols, higher priority *)
74 |   (* compared to the identifiers, meaning that there can't be aliases *)
75 |   (* or interfaces called with a protocol name *)
76 |   | "tcp"                        { TCP }
77 |   | "udp"                        { UDP }
78 |   | "icmp"                       { ICMP }
79 |   (* Identifiers and IPs *)
80 |   | identifier as value          { IDENTIFIER value }
81 |   | additional as value          { FORMULA (clean value '|') }
82 |   | "*"                          { STAR }
83 |   | "@"                          { AT }
84 |   | net_addr  as value           { NET_IP value }
85 |   | host_addr  as value          { HOST_IP value }
86 |   (* Ports are not always specified, so we manage them with a specific rule *)
87 |   | port as value                { PORT (int_of_string (clean value ':')) }
88 |   (* Operators for firewall policies *)
89 |   | ">"                          { ALLOW }
90 |   | "<>"                         { TWALLOW }
91 |   | "/"                          { DROP }
92 |   | "//"                         { REJECT }
93 |   (* Brackets are used to define SNATs and DNATs togehter with the char *)
94 |   (*  '.' used when masquerade is requested *)
95 |   | "["                          { LBRACK }
96 |   | "."                          { DOT }
97 |   | "]"                          { RBRACK }
98 | (* Comment entrypoint: used to ignore comments in the main conf part *)
99 | and comment state_ref = parse
100|   | "\n"                         { main state_ref lexbuf }
101|   | eof                          { EOF }
102|   | _                            { comment state_ref lexbuf }
103| (* Ccomment entrypoint: used to ignore comments in the Custom rules part *)
104| and ccomment state_ref = parse
105|   | "\n"                         { cstrule state_ref lexbuf }
106|   | eof                          { EOF }
107|   | _                            { ccomment state_ref lexbuf }
108| (* Cstrule entrypoint: used to recognize the custom rules *)
109| and cstrule state_ref = parse
110|   | "OPTIONS"                    { state_ref := MainConf; OPTION }
111|   | "\n"                         { cstrule state_ref lexbuf }
112|   | '#'                          { ccomment state_ref lexbuf }
```

```
113 |   | custom as value              { CUSTOMRULE value }
114 |   | eof                          { EOF }
115 |
116 | {
117 |   (* Current state is MainConf *)
118 |   let current = ref MainConf;;
119 |
120 |   (* To get the next token, the current state is matched and decisions are *)
121 |   (* taken *)
122 |   let next_token =
123 |     (fun lexbuf -> match !current with
124 |      | MainConf -> main current lexbuf
125 |      | CustomRules -> cstrule current lexbuf)
126 | ;;
127 | }
```

Listing A.2: File parser.mly

```
%{
  open Mast
%}

/* Main key words */
%token OPTION INTERFACE ALIAS FIREWALL POLICY CUSTOM LOCAL STAR AT
/* Ids, addresses and ports */
%token <string> FORMULA
%token <string> CUSTOMRULE
%token <string> IDENTIFIER
%token <string> NET_IP
%token <string> HOST_IP
%token <int> PORT
/* Operators */
%token ALLOW DROP REJECT TWALLOW
/* Brackets and dots */
%token LBRACK DOT RBRACK
/* Protocols */
%token TCP UDP ICMP
/* EOF */
%token EOF

%start config
%type <Mast.config> config

%%
/* The options part always begins with the keyword OPTION */
Options:
  OPTION OptionList                   { $2 }
;
/* The list of options is made of two IDs, one for the keyword, one for the */
/* value */
OptionList:
  IDENTIFIER IDENTIFIER OptionList    { Option($1, $2)::$3 }
 |                                    { [] }
;
/* The interface part begins with the keyword INTERFACE */
InterfaceDecl:
  INTERFACE InterfList                { $2 }
;
/* This is the structure of an interface declaration */
InterfList:
  IDENTIFIER IDENTIFIER NET_IP  InterfList
                                      { Interface($1, $2, $3)::$4 }
 |                                    { [] } /* Empty list */
;
/* The alias part begins with the keyword ALIAS */
AliasDecl:
  ALIAS AliasList                     { $2 }
;
/* This is how a new alias is declared */
AliasList:
  IDENTIFIER HOST_IP AliasList        { Hostalias($1, $2)::$3 }
 |IDENTIFIER NET_IP AliasList         { Netalias($1, $2)::$3 }
 |                                    { [] }
;
/* Firewall rules are specified after the keyword FIREWALL */
FirewallConf:
  FIREWALL FirewList                  { $2 }
;
```

```
/* Firewall rules have a rather complex structure: */
/* Each of them has two endpoints, possibly two nats (snat and dnat) */
/* an operator and a protocol. Some of them could not be specified but in the */
/* AST they are all explicit */
FirewList:
  Endp Nt ALLOW Nt Endp Prtc Formula FirewList
                                    { Allow($1, $2, $4, $5, $6, $7)::$8 }
 |Endp Nt TWALLOW Nt Endp Prtc Formula FirewList
                                    { Twallow($1, $2, $4, $5, $6, $7)::$8 }
 |Endp Nt DROP Nt Endp Prtc Formula FirewList
                                    { Drop($1, $2, $4, $5, $6, $7)::$8 }
 |Endp Nt REJECT Nt Endp Prtc Formula FirewList
                                    { Reject($1, $2, $4, $5, $6, $7)::$8 }
 |                                  { [] }
;
/* This is how an endpoint looks like */
Endp:
  IDENTIFIER If Prt                 { Name($1, $2, $3) }
 |HOST_IP If Prt                    { Ip("h-" ^ $1, $2, $3) }
 |NET_IP If Prt                     { Ip("n-" ^ $1, $2, $3) }
 |LOCAL Prt                         { Local($2) }
 |STAR                              { Star }
;
/* Basically if no port is explicitly specified, we use the dummy 0 value */
Prt:
  PORT                              { $1 }
 |                                  { 0 }
;
/* Interface management for Mignis+ */
If:
  AT IDENTIFIER                     { If($2) }
 |                                  { Noif }
;
/* Nat, managing the masquerade case */
Nt:
  LBRACK DOT RBRACK                 { Masquerade }
 |LBRACK Endp RBRACK                { Nat($2) }
 |                                  { Nonat } /* No nat is specified */
;
/* The protocol */
Prtc:
  TCP                               { Tcp }
 |UDP                               { Udp }
 |ICMP                              { Icmp }
 |                                  { Noprotocol } /* no explicit protocol */
;
/* Formulas in mignis+ */
Formula:
  FORMULA                           { Formula($1) }
 |                                  { Noformula }
/* The policy section begins with the keyword POLICY */
PolicyConf:
  POLICY PolicyList                 { $2 }
;
/* This is the structure of a default policy rule */
PolicyList:
  Endp DROP Endp Prtc PolicyList    { Default(Pdrop, $1, $3, $4)::$5 }
 |Endp REJECT Endp Prtc PolicyList  { Default(Preject, $1, $3, $4)::$5 }
 |                                  { [] }
;
/* The custom rules section begins with the keyword CUSTOM */
CustomRules:
```

```
123 ||   CUSTOM CustomList                  { $2 }
124 || ;
125 || /* All the custom rules are added to a specific list */
126 || CustomList:
127 ||   CUSTOMRULE CustomList               { $1::$2 }
128 ||  |                                   { [] }
129 || ;
130 || /* Main configuration */
131 || config:
132 ||   Options InterfaceDecl AliasDecl FirewallConf PolicyConf CustomRules config
133 ||                                       { Firewall($1, $2, $3, $4, $5, $6)::$7 }
134 ||  |EOF                                 { [] }
135 || ;
136 ||
137 || %%
```

Listing A.3: File mast.mli

```
type config           =  firewall list

and firewall          =  Firewall of (option list) *
                                     (interface list) *
                                     (alias list) *
                                     (rule list) *
                                     (policy list) *
                                     (custom list)

and option            = Option of id * id

and  interface        = Interface of id * id * net_ip

and alias             = Hostalias of id * host_ip
                      | Netalias of id * net_ip

and rule              = op

and policy            = Default of pop * endpoint * endpoint * protocol

and custom            = crule

and endpoint          = Name of id * ifc * port
                      | Ip of host_ip * ifc * port
                      | Local of port
                      | Star

and ifc               = Noif
                      | If of id

and nat               = Nat of endpoint
                      | Masquerade
                      | Nonat

and op                = Allow of endpoint * nat *
                                 nat * endpoint *
                                 protocol *
                                 formula
                      | Twallow of endpoint * nat *
                                   nat * endpoint *
                                   protocol *
                                   formula
                      | Drop of endpoint * nat *
                                nat * endpoint *
                                protocol *
                                formula
                      | Reject of endpoint * nat *
                                  nat * endpoint *
                                  protocol *
                                  formula

and pop               = Pdrop
                      | Preject

and protocol          = Tcp
                      | Udp
                      | Icmp
                      | Noprotocol

and formula           = Formula of crule
```

```
61 ||                       | Noformula
62 ||
63 || and id = string
64 || and crule = string
65 || and net_ip = string
66 || and host_ip = string
67 || and port = int
```

Listing A.4: File scope.ml

```
1
2  open Lexer;;
3  open Parser;;
4  open List;;
5
6  (* Definition of several list: each of them keeps track of a certain part of *)
7  (* the Mignis Abstract Syntax Tree. For all list, the n-th position describes *)
8  (* the configuration of the n-th firewall *)
9
10 (* Options *)
11 let option_list:((Mast.id * Mast.id) list) list ref = ref [];;
12 (* Interfaces *)
13 let interface_list:((Mast.id * Mast.id * Mast.net_ip) list) list ref = ref [];;
14 (* Aliases *)
15 let alias_list:((Mast.id * string) list) list ref = ref [];;
16 (* Firewall rules *)
17 let rule_list:(Mast.op list) list ref = ref [];;
18 (* Policies *)
19 let policy_list:(Mast.policy list) list ref = ref [];;
20 (* Custom rules *)
21 let crule_list:(Mast.crule list) list ref = ref [];;
22
23 (* Aux function to read from a source file *)
24 let read_source file = Lexing.from_channel (open_in file);;
25
26 (* Aux func to lex and parse a configuration file *)
27 let lex_and_parse file =
28   try config next_token (read_source file) with
29   | Failure error -> failwith ("Lexer error: " ^ error)
30   | Parsing.Parse_error -> failwith ("Parse error")
31 ;;
32
33 (* Aux func to create the option structure *)
34 let rec create_option (opt:Mast.option list) =
35   match opt with
36   | Mast.Option(keyword,value)::rest    -> (keyword,value)::create_option rest
37   | []                                  -> []
38 ;;
39
40 (* Aux func to create the interface alias and binding structure *)
41 let rec create_interface (ifs:Mast.interface list) =
42   match ifs with
43   | Mast.Interface(name,noi,nip)::rest  -> (name,noi,nip)::create_interface rest
44   | []                                  -> []
45 ;;
46
47 (* Aux func to create the alias table *)
48 let rec create_alias (als:Mast.alias list) =
49   match als with
50   | Mast.Hostalias(name,ip)::rest       -> (name,"h-" ^ ip)::create_alias rest
51   | Mast.Netalias(name,nip)::rest       -> (name,"n-" ^ nip)::create_alias rest
52   | []                                  -> []
53 ;;
54
55 (* This function create a complete firewall structure *)
56 let conf_firewall (fw:Mast.firewall) =
57   let Mast.Firewall(options,interfaces,aliases,rules,policies,custom) = fw in
58   option_list := !option_list @ [(create_option options)];
59   interface_list := !interface_list @ [(create_interface interfaces)];
60   alias_list := !alias_list @ [(create_alias aliases)];
```

```
  rule_list := !rule_list @ [rules];
  policy_list := !policy_list @ [policies];
  crule_list := !crule_list @ [custom]
;;

(* This function manages all the firewalls structures *)
let rec create_conf_table (ast:Mast.config) =
  match ast with
  | fw::rest                          -> conf_firewall fw;
                                         create_conf_table rest
  | []                                -> ()
;;

(* Main function: it creates the complete configuration table *)
let start source =
  let ast = lex_and_parse source in
  option_list := [];
  interface_list := [];
  alias_list := [];
  rule_list := [];
  policy_list := [];
  crule_list := [];
  create_conf_table ast
;;
```

Listing A.5: File compiler.ml

```
(* List of string in which we save the compiled configuration for each *)
(* firewall *)
let compiled:(string list) ref = ref [];;

(* List of operands to keep track of overlapping rules *)
let overlaps:(string list) ref = ref [];;

(* Are we on a Mignis or Mignis+ configuration file? *)
type conf_t = Mignis | MignisPlus | NotSet;;
let conf:(conf_t) ref = ref NotSet;;

(* This global expression is used to tokenize the operands. It's global *)
(* because we need it in more than one function and for since the overlaps *)
(* detection is not efficient, we don't want to call the Str.regexp more *)
(* than once*)
let comma = Str.regexp ";";;

(* This global expression is used only to pass a message in case of bad rules *)
let warning:string ref = ref "";;

(* Aux function that is used to create the option string *)
let rec create_options opt =
  match opt with
  | (keyword,value)::rest        ->
    "OPTN:" ^ keyword ^ ";" ^ value ^ "\n" ^ create_options rest
  | []                           -> ""
;;

(* Aux function that is used to create the bounds between interfaces and *)
(* network ip addresses *)
let rec create_interfaces ifs =
  match ifs with
  | (name,nic,ip)::rest          ->
    "BIND:" ^ nic ^ ";" ^ ip ^ "\n" ^ create_interfaces rest
  | []                           -> ""
;;

(* The following two aux functions are used to find an alias or an interface *)
(* Note that interfaces are bound to a net ip address, so here the nic name *)
(* is returned, while aliases are never actually saved: the corresponding ip *)
(* is returned *)
let rec find_alias lst needle =
  match lst with
  | (name,ip)::rest              ->
    if name = needle then
      ip
    else
      find_alias rest needle
  |  []                          -> ""
;;

let rec find_interface lst needle =
  match lst with
  | (name,nic,ip)::rest          ->
    if name = needle then
      nic
    else
      find_interface rest needle
  |  []                          -> ""
;;
```

```
61
62  (* Aux function used to find an alias or an interface *)
63  (* Please note that aliases have a higher priority *)
64  let rec find needle ambient =
65    let alias = List.nth !Scope.alias_list ambient in
66    let result_alias = find_alias alias needle in
67    if result_alias = "" then
68      let interface = List.nth !Scope.interface_list ambient in
69      find_interface interface needle
70    else
71      result_alias
72  ;;
73
74  (* This function is the one that is allowed to track if we are on a Mignis *)
75  (* or Mignis+ configuration file and it keeps things coherent *)
76  let set_interface inf amb nat =
77    match inf with
78    | Mast.Noif                    ->
79      if !conf = MignisPlus && nat = false then
80        failwith("Mignis and Mignis+ rules cannot be used together")
81      else if !conf = NotSet then
82        begin
83          conf := Mignis;
84          ""
85        end
86      else
87        ""
88    | Mast.If(id)                  ->
89      let resolved = find id amb in
90      if resolved = "" then
91        failwith("Interface not declared")
92      else
93        if !conf = Mignis && nat = false then
94          failwith("Mignis and Mignis+ rules cannot be used together")
95        else if !conf = NotSet then
96          begin
97            conf := MignisPlus;
98            resolved
99          end
100       else
101         resolved
102 ;;
103
104 (* Aux functions to correctly compile all the components of a rule *)
105 let set_endpoint e amb nat=
106   match e with
107   | Mast.Name(host,interface,port)
108                                  ->
109     let resolved = find host amb in
110     if resolved = "" then
111       failwith("Alias or interface not declared")
112     else
113       resolved ^ ";" ^ (set_interface interface amb nat) ^ ";" ^
114         string_of_int(port)
115   | Mast.Ip(host,interface,port)
116                                  ->
117     host ^ ";" ^ (set_interface interface amb nat) ^ ";" ^ string_of_int(port)
118   | Mast.Local(port)             ->
119     "LOCAL" ^ ";;" ^ string_of_int(port)
120   | Mast.Star                    -> "ANY;;0"
121 ;;
122
```

```
123 let set_nat n amb =
124   match n with
125   | Mast.Nat(ep)                -> set_endpoint ep amb true
126   | Mast.Masquerade             -> "MASQUERADE;;0"
127   | Mast.Nonat                  -> ";;0"
128 ;;
129
130 let set_protocol p =
131   match p with
132   | Mast.Tcp                    -> "TCP"
133   | Mast.Udp                    -> "UDP"
134   | Mast.Icmp                   -> "ICMP"
135   | Mast.Noprotocol             -> "ANY"
136 ;;
137
138 let set_formula f =
139   match f with
140   | Mast.Formula(f)             -> f
141   | Mast.Noformula              -> ""
142 ;;
143
144 (* Aux function used to create the operands of a rule *)
145 let set_op se sn dn de pr fr amb =
146   (set_endpoint se amb false) ^ ";" ^
147   (set_nat sn amb) ^ ";" ^
148   (set_endpoint de amb false) ^ ";" ^
149   (set_nat dn amb) ^ ";" ^
150   (set_protocol pr) ^ ";" ^
151   (set_formula fr)
152 ;;
153
154 (* Aux function to check if a string is a substring of another *)
155 (* Source code found at *)
156 (* http://stackoverflow.com/questions/11193783/ocaml-strings-and-substrings *)
157 (* Last checked october 4th 2016 *)
158 let contains s1 s2 =
159   let re = Str.regexp_string s2 in
160   try
161     ignore (Str.search_forward re s1 0);
162     true
163   with Not_found -> false
164
165 (* Aux function to find overlaps in two rules *)
166 let chk_rule r1 r2 =
167   let s1 = List.nth r1 0 ^ "," ^ List.nth r1 1 ^ "," ^ List.nth r1 2 in
168   let s2 = List.nth r2 0 ^ "," ^ List.nth r2 1 ^ "," ^ List.nth r2 2 in
169   let sn1 = List.nth r1 3 ^ "," ^ List.nth r1 4 ^ "," ^ List.nth r1 5 in
170   let sn2 = List.nth r2 3 ^ "," ^ List.nth r2 4 ^ "," ^ List.nth r2 5 in
171   let d1 = List.nth r1 6 ^ "," ^ List.nth r1 7 ^ "," ^ List.nth r1 8 in
172   let d2 = List.nth r2 6 ^ "," ^ List.nth r2 7 ^ "," ^ List.nth r2 8 in
173   let dn1 = List.nth r1 9 ^ "," ^ List.nth r1 10 ^ "," ^ List.nth r1 11 in
174   let dn2 = List.nth r2 9 ^ "," ^ List.nth r2 10 ^ "," ^ List.nth r2 11 in
175   if s1 = s2 && d1 = d2 &&
176      sn1 = sn2 && sn1 = ",,0" &&
177      dn1 = dn2 && dn1 = ",,0" then
178     begin
179       warning := !warning ^
180         "A rule with same source and destination has been found\n";
181       true
182     end
183   else if s1 = s2 && d1 = d2 &&
184           dn1 <> dn2 &&
```

```
185           (dn1 = ",,0" || dn2 = ",,0") &&
186           sn1 = sn2 then
187     begin
188       warning := !warning ^ "A rule with dNAT is overlapping a generic rule\n";
189       true
190     end
191   else if s1 = s2 && d1 <> d2 &&
192           dn1 = dn2 && dn1 <> ",,0" then
193     begin
194       warning := !warning ^ "A rule with an overlapping dNAT has been found\n";
195       true
196     end
197   else if s1 = s2 &&
198           d1 <> d2 &&
199           (d1 = "ANY,,0" || d2 = "ANY,,0") then
200     begin
201       warning := !warning ^ "A rule is overlapping a * destination\n";
202       true
203     end
204   else if sn1 <> ",,0" && dn1 <> ",,0" then
205     begin
206       warning := "A rule cannot specify a sNAT and a dNAT at the same time";
207       false
208     end
209   else if dn1 = "MASQUERADE,,0" then
210     begin
211       warning := "Destination masquarade is not allowed";
212       false
213     end
214   else if sn1 = "ANY,,0" || dn1 = "ANY,,0" then
215     begin
216       warning := "Wildcard * cannot be used in NAT declarations";
217       false
218     end
219   else if (contains sn1 "LOCAL") || (contains dn1 "LOCAL") then
220     begin
221       warning := "Keyword local cannot be used in NAT declarations";
222       false
223     end
224   else
225     true
226 ;;
227
228
229 (* Function to check whether there are overlaps or not *)
230 let rec check_overlaps opns current =
231   match current with
232   | rule::rest                    -> let rule_token = Str.split comma rule in
233                                      if chk_rule opns rule_token = false then
234                                        false
235                                      else
236                                        check_overlaps opns rest
237   | []                            -> true
238 ;;
239
240 (* Aux function used to create the list of rules *)
241 let rec create_rules rls ambient =
242   warning := "";
243   match rls with
244   | Mast.Allow(from,snat,dnat,dest,prt,frm)::rest
245                                   -> let operands =
246                                        set_op from snat dnat dest prt frm ambient
```

```
                                                                    in
                                  if check_overlaps (Str.split comma operands)
                                    !overlaps = true then
                                    begin
                                      overlaps := !overlaps @ [operands];
                                      if !warning <> "" then
                                        Printf.printf "Warning: %s" !warning;
                                      (create_rules rest ambient) ^
                                      "ALLW:" ^
                                      operands ^
                                      "\n"
                                    end
                                  else
                                    failwith(!warning)
  | Mast.Twallow(from,snat,dnat,dest,prt,frm)::rest
                                -> let operands =
                                     set_op from snat dnat dest prt frm ambient
                                                                    in
                                   if check_overlaps (Str.split comma operands)
                                     !overlaps = true then
                                     begin
                                       overlaps := !overlaps @ [operands];
                                       if !warning <> "" then
                                         Printf.printf "Warning: %s" !warning;
                                       (create_rules rest ambient) ^
                                       "TALW:" ^
                                       operands ^
                                       "\n"
                                     end
                                   else
                                     failwith(!warning)
  | Mast.Drop(from,snat,dnat,dest,prt,frm)::rest
                                -> if !conf = MignisPlus then
                                     failwith("Mignis+ does not allow " ^
                                              "for negative rules");
                                   let operands =
                                     set_op from snat dnat dest prt frm ambient
                                                                    in
                                   if check_overlaps (Str.split comma operands)
                                     !overlaps = true then
                                     begin
                                       overlaps := !overlaps @ [operands];
                                       if !warning <> "" then
                                         Printf.printf "Warning: %s" !warning;
                                      "DROP:" ^
                                      operands ^
                                      "\n" ^ (create_rules rest ambient)
                                     end
                                  else
                                    failwith(!warning);
  | Mast.Reject(from,snat,dnat,dest,prt,frm)::rest
                                -> if !conf = MignisPlus then
                                     failwith("Mignis+ does not allow " ^
                                              "for negative rules");
                                   let operands =
                                     set_op from snat dnat dest prt frm ambient
                                                                    in
                                   if check_overlaps (Str.split comma operands)
                                     !overlaps = true then
                                     begin
                                       overlaps := !overlaps @ [operands];
                                       if !warning <> "" then
```

```
                                        Printf.printf "Warning: %s" !warning;
                                      "RJCT:" ^
                                      operands ^
                                      "\n" ^ (create_rules rest ambient)
                                    end
                                  else
                                    failwith(!warning);
  | []                          -> ""
;;

let set_policy_rule r =
  match r with
  | Mast.Pdrop                  -> "PDRP:"
  | Mast.Preject                -> "PRJC:"
;;

(* Aux function used to create the policy rules *)
let rec create_policies pol amb =
  match pol with
  | Mast.Default(op,end1,end2,pr)::rest
                                -> (set_policy_rule op) ^
                                   (set_endpoint end1 amb false) ^ ";" ^
                                   (set_endpoint end2 amb false) ^ ";" ^
                                   (set_protocol pr) ^ "\n" ^
                                   create_policies rest amb
  | []                          -> ""
;;

(* Aux function to include all the custom rules *)
let rec create_custom cstm =
  match cstm with
  | str::rest                   -> "CSTM:" ^ str ^ "\n" ^ create_custom rest
  | []                          -> ""
;;

(* Aux function that is used to actually compile the configurations *)
let rec build_conf len =
  if len > 0 then
    begin
      let current_option = List.nth !Scope.option_list (len - 1) in
      let current_interface = List.nth !Scope.interface_list (len - 1) in
      let current_rules = List.nth !Scope.rule_list (len - 1) in
      let current_policies = List.nth !Scope.policy_list (len - 1) in
      let current_custom = List.nth !Scope.crule_list (len - 1) in
      let option_string = create_options current_option in
      let interface_string = create_interfaces current_interface in
      overlaps := [];
      let rules_string = create_rules current_rules (len - 1) in
      let policies_string = create_policies current_policies (len - 1) in
      let custom_string = create_custom current_custom in
      let all = option_string ^
                interface_string ^
                rules_string ^
                policies_string ^
                custom_string in
      compiled := all::(!compiled);
      build_conf (len - 1)
    end
  else
    if len < 0 then
      failwith("General error")
    else
```

```
371       ()
372 ;;
373
374 (* Main function of the backend component. It just call the build_conf *)
375 (* with the correct length. Please note that all the lists in the Scope *)
376 (* module have the same length *)
377 let compile file =
378   Scope.start file;
379   build_conf (List.length !Scope.option_list)
380 ;;
```

Listing A.6: File mignis.ml

```
open Printf;;

let forced = ref false;;

let rec write_to_file comp_conf prog dir =
  match comp_conf with
  | str::rest        ->
    let filename = dir ^ "fw" ^ (string_of_int prog) ^ ".config" in
    let outfile = open_out filename in
    fprintf outfile "%s" str;
    printf "Configuration file %s successfully created\n" filename;
    write_to_file rest (prog + 1) dir
  | []               -> ()
;;


let par_len = Array.length (Sys.argv) in
if par_len < 2 || par_len > 3 then
  begin
    printf "Usage: ./mignis [-f] <file_name>\n";
    exit 0
  end
else
  begin
    for i = 1 to par_len - 2 do
      if Sys.argv.(i) = "-f" then
        forced := true
      else
        begin
          printf "Unrecognized option: %s\n" Sys.argv.(i);
          exit 0
        end
    done;
    let space = Str.regexp " " in
    let name_escaped = Str.global_replace space "\\ " Sys.argv.(par_len - 1) in
    if Sys.file_exists name_escaped then
      begin
        let dest_dir = (Filename.dirname name_escaped ^
                        "/compiled/") in
        if Sys.file_exists dest_dir then
          begin
            if !forced = false then
              begin
                printf "Directory %s already exists. Use -f to overwrite\n"
                       dest_dir;
                exit 0
              end
            else
              let _ = Sys.command ("rm -Rf " ^ dest_dir ^ "*") in
              let _ = Sys.command ("rmdir " ^ dest_dir) in
              printf "Directory %s has been deleted\n" dest_dir;
          end
        else ();

        let _ = Sys.command ("mkdir " ^ dest_dir) in
        printf "Directory %s created\n" dest_dir;
        Compiler.compile name_escaped;
        write_to_file !Compiler.compiled 0 dest_dir
      end
    else
```

```
61 ||       printf "File %s does not exist\n" name_escaped
62 ||   end
63 || ;;
```

B

COMPLETE SOURCE CODE FOR THE FINAL TRANSLATOR (FRONTEND COMPONENT)

Listing B.1: File generic_engine.py

```
__author__ = "Alessio Zennaro"

from abc import ABCMeta, abstractmethod
import os
import itertools

''' This class is used as a model for all target languages.
    Basically it is an abstract class that is able to read all the files written
    in the intermediate representation and, from those files, it produces the
    final configuration in th target language. Since this is a generic model,
    the translation into the final language is kept abstract
'''
class GenericEngine:
    __metaclass__ = ABCMeta

    ''' Following there are variables that represent the keywords of the
        intermediate language. These are useful in order to avoid to use direct
        strings in the code
    '''
    OPTN = "OPTN"
    BIND = "BIND"
    ALLW = "ALLW"
    DROP = "DROP"
    RJCT = "RJCT"
    TALW = "TALW"
    PDRP = "PDRP"
    PRJC = "PRJC"
    CSTM = "CSTM"

    LOCAL = "LOCAL"
    ANY = "ANY"
    MASQUERADE = "MASQUERADE"

    ''' This method just inits the language property '''
    def __init__(self, directory):
        self.language = ""
        self.directory = directory


    ''' This method is used to actually read all the intermediate
        representations and then translate them into the final target language
        via the translate method.
        The ../final folder is supposed to already exist.
        The files inside the ../final folder are look like the following
        fw<index>.<target_language>
        If an IOErr occurs, the current file is skipped.
        It returns the number of final configurations written to disk
    '''
    def compile(self):
        # The complete file name structure is
```

```
51          # ../final/fw<index>.<targe_language>
52          prefix = self.directory + "final/fw"
53          suffix = "." + self.language
54
55          # We get all the configurations written in the intermediate
56          # representation
57          conf_list = self.read_files()
58
59          n = 0
60          # For all the configurations found
61          for conf in conf_list:
62              final_conf = self.translate(conf)  # Translate it
63              file_name = prefix + str(n) + suffix  # Create the file name
64              try:
65                  # Create a new file and write the final configuration in it
66                  out_stream = open(file_name, "w")
67                  out_stream.write(final_conf)
68                  out_stream.flush()
69                  out_stream.close()
70                  n += 1
71              except IOError as _:
72                  # If something goes wrong, skip the file and continue with
73                  # the next
74                  print("ERR: Skipping output file %s because of an I/O error"
75                        % file_name)
76                  continue
77
78          # Return the number of final configurations written
79          return n
80
81
82      ''' This method is used to read all the configuration files
83          written in the intermediate mignis representation.
84          Files must be in the ../compiled folder and file names must
85          comply with the following naming convention
86          fw<index>.config
87          If a file is not readable then it is skipped.
88          If a file does not exist, the procedure terminates
89      '''
90      def read_files(self):
91          # The complete file name structure is ../compiled/fw<index>.config
92          prefix = self.directory + "compiled/fw"
93          suffix = ".config"
94
95          # The returned list is made of strings and each string is a
96          # configuration
97          conf_list = []
98          total = 0
99
100         # Forever...
101         for i in itertools.count():
102             file_name = prefix + str(i) + suffix  # The complete name is built
103             if os.path.isfile(file_name):  # Does the file exist?
104                 try:
105                     # If it exists (and it is readable): open it, read it and
106                     # close it
107                     in_stream = open(file_name, "r")
108                     # Add the content to the list
109                     conf_list.append(in_stream.read())
110                     in_stream.close()
111                     total += 1
112                 except IOError as _:
```

```
113                     # If it exists but something goes wrong: skip it and
114                     # inform user
115                     print("ERR: Skipping input file %s since it isn't readable"
116                           % file_name)
117             else:
118                 # If it doesn't exist: break the loop
119                 break
120
121         # Inform the user of the number of configurations successfully read and
122         # return the list
123         print("\nINF: Successfully read %d files\n" % total)
124         return conf_list
125
126     ''' This method is used to parse a line. A line is made of a string
127         (of 4 chars), a colon and then another string made of a sequence of
128         string separated by a semicolon. It returns a tuple of two elements:
129         the first one is the first string, the second one is a list of all
130         the parameters found in the second string
131     '''
132     def parse_line(self, line):
133         cmd_par = line[0:4]
134         par_list = line[5:].split(';')
135
136         return (cmd_par, par_list)
137
138     ''' This method is used to get all the details of a rule, grouped by their
139         meaning. For instance the first three elements are to identify the
140         source endpoint, so they are grouped together (host, int, port).
141         We are sure that all these elements exist because the intermediate
142         representation provides all the infos even if they are empty.
143     '''
144     def get_rule_details(self, par):
145         if len(par) != 14:  # Here something is wrong, not a rule
146             print("ERR: bad parameter")
147             exit(-1)  # We must exit, unrecoverable error!
148
149         # source endpoint: (host, interface, port)
150         source = (par[0], par[1], par[2])
151         # sNAT: from a syntactic POV it is an endpoint
152         snat = (par[3], par[4], par[5])
153         # destination endpoint
154         destination = (par[6], par[7], par[8])
155         # dNAT
156         dnat = (par[9], par[10], par[11])
157         # protocol
158         protocol = par[12]
159         # custom rules
160         formula = par[13]
161
162         # Return a list with all these infos in tuples
163         return [source, snat, destination, dnat, protocol, formula]
164
165     ''' This method is used to modify a string in order to change a parameter.
166         This is particularly useful when the two way allow operator (<>) is
167         used: for instance the source port in one rule becomes the destination
168         port in the twin rule. By simply using this method with the
169         correct parameters, the task of updating the rule is very easy
170     '''
171     def switch_elements(self, string, pattern_list):
172         return_string = string
173         for pattern in pattern_list:
174             return_string = return_string.replace(pattern[0], pattern[1])
```

```
175
176             return return_string
177
178         ''' This abstract method is the very heart of the whole program. It takes
179             a cofiguration written in intermediate representation and it translates
180             it into a configuration written in the target language.
181         '''
182         @abstractmethod
183         def translate(self, configuration): pass
```

Listing B.2: File example_engine.py

```
__author__ = "Alessio Zennaro"

from generic_engine import GenericEngine


''' This class extends the GenericEngine and can be used as a template class
    for new supported target languages.
    It consists of just two methods: the constructor and the translate method:
     * The constructor is used only to specify the target language we are
       implementing
     * The translate method is the one that actually translate the intermediate
       representation into the final target language. Needless to say, this is
       the method that must perform the most important operations
'''
class ExampleEngine(GenericEngine):

    ''' Constructor method. It just sets the name of the language and the
        directory
    '''
    def __init__(self, directory):
        GenericEngine.__init__(self, directory)
        self.language = "example"  # This is just an example

    ''' translate method. It performs the whole translation from the
        intermediate representation into the target language
    '''
    def translate(self, configuration):
        # Again, this is just an example
        return "This is just an example of final target language!"
```

Listing B.3: File netfilter_engine.py

```
__author__ = "Alessio Zennaro"

from generic_engine import GenericEngine

''' This class allows for the translation from mignis to netfilter/iptables
    via the intermediate representation.
    See example_engine.py file for full documentation regarding the structure of
    the file/class.
'''


class NetfilterEngine(GenericEngine):

    # Flag for the Mignis+ warning
    migplus = False

    # Here all the iptables rule templates are defined in variables, to avoid
    # the use of strings in the code.
    BASIC_FILTER = "*filter\n" + \
                   "-P INPUT DROP\n" + \
                   "-P FORWARD DROP\n" + \
                   "-P OUTPUT DROP\n{0}"
    BASIC_MANGLE = "*mangle\n" + \
                   "-P PREROUTING DROP\n"
    MANGLE_LO = "-A PREROUTING -i lo -j ACCEPT\n"
    BASIC_NAT = "*nat\n"
    DEFAULT_ESTABLISHED = \
        "-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT\n" + \
        "-A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT\n" + \
        "-A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT\n{0}"
    DEFAULT_FILTER = \
        "-A INPUT -i lo -j ACCEPT -m comment " + \
          "--comment \"loopback (default rules)\"\n" + \
        "-A INPUT -d 255.255.255.255 -j ACCEPT -m comment " + \
          "--comment \"broadcast (default r.)\"\n" + \
        "-A INPUT -d 224.0.0.0/4 -j ACCEPT -m comment " + \
          "--comment \"multicast (default r.)\"\n"
    DEFAULT_MANGLE = \
        "-A PREROUTING -m state --state INVALID,UNTRACKED -j DROP " + \
          "-m comment --comment \"inv. def.\"\n" + \
        "-A PREROUTING -d 255.255.255.255 -j ACCEPT -m comment " + \
          "--comment \"default r.\"\n" + \
        "-A PREROUTING -d 224.0.0.0/4 -j ACCEPT -m comment " + \
          "--comment \"default r.\"\n"
    LOGGING_FILTER = "-N filter_drop\n" + \
                     "-N filter_drop_icmp" + \
                     "-A filter_drop_icmp -j LOG --log-prefix \"DROP-icmp \"\n" + \
                     "-A filter_drop_icmp -j DROP\n" + \
                     "-A filter_drop -p icmp -j filter_drop_icmp\n" + \
                     "-N filter_drop_udp\n" + \
                     "-A filter_drop_udp -j LOG --log-prefix \"DROP-udp \"\n" + \
                     "-A filter_drop_udp -j DROP\n" + \
                     "-A filter_drop -p udp -j filter_drop_udp\n" + \
                     "-N filter_drop_tcp\n" + \
                     "-A filter_drop_tcp -j LOG --log-prefix \"DROP-tcp \"\n" + \
                     "-A filter_drop_tcp -j DROP\n" + \
                     "-A filter_drop -p tcp -j filter_drop_tcp\n" + \
                     "-A filter_drop -j LOG --log-prefix \"DROP-UNK \"\n" + \
                     "-A filter_drop -j DROP\n" + \
                     "-A INPUT -j filter_drop\n" + \
```

```
61                      "-A OUTPUT -j filter_drop\n" + \
62                      "-A FORWARD -j filter_drop\n"
63      LOGGING_MANGLE = "-N mangle_drop\n" + \
64                      "-N mangle_drop_icmp\n" + \
65                      "-A mangle_drop_icmp -j LOG " + \
66                        "--log-prefix \"MANGLE-DROP-ICMP \"\n" + \
67                      "-A mangle_drop_icmp -j DROP\n" + \
68                      "-A mangle_drop -p icmp -j mangle_drop_icmp\n" + \
69                      "-N mangle_drop_udp\n" + \
70                      "-A mangle_drop_udp -j LOG " + \
71                        "--log-prefix \"MANGLE-DROP-UDP \"\n" + \
72                      "-A mangle_drop_udp -j DROP\n" + \
73                      "-A mangle_drop -p udp -j mangle_drop_udp\n" + \
74                      "-N mangle_drop_tcp\n" + \
75                      "-A mangle_drop_tcp -j LOG " + \
76                        "--log-prefix \"MANGLE-DROP-TCP \"\n" + \
77                      "-A mangle_drop_tcp -j DROP\n" + \
78                      "-A mangle_drop -p tcp -j mangle_drop_tcp\n" + \
79                      "-A mangle_drop -j LOG " + \
80                        "--log-prefix \"MANGLE-DROP-UNK \"\n" + \
81                      "-A mangle_drop -j DROP\n" + \
82                      "-A PREROUTING -j mangle_drop\n"
83
84      BIND_ACCEPT = \
85          "-A PREROUTING -i {0} -s {1} -j ACCEPT -m comment --comment \"{2}\"\n"
86      BIND_ANY_ACCEPT = \
87          "-A PREROUTING -i {0} -j ACCEPT -m comment --comment \"{1}\"\n"
88      BIND_ANY_DROP = \
89          "-A PREROUTING -i {0} -s {1} -j DROP -m comment --comment \"{2}\"\n"
90      MANGLE_NAT = \
91          "-A PREROUTING {0} {1} {2} {3} {4} -m state --state NEW -j DROP " + \
92            "-m comment --comment \"{5}\"\n"
93      RULE_FWD = \
94          "-A FORWARD {0} {1} {2} {3} {4} {5} -j {6} " + \
95            "-m comment --comment \"{7}\"\n"
96      RULE_IN = \
97          "-A INPUT {0} {1} {2} {3} {4} -j {5} -m comment --comment \"{6}\"\n"
98      RULE_OUT = \
99          "-A OUTPUT {0} {1} {2} {3} {4} -j {5} -m comment --comment \"{6}\"\n"
100     RULE_MASQUERADE = \
101         "-A POSTROUTING {0} {1} {2} {3} {4} -j MASQUERADE " + \
102           "-m comment --comment \"{5}\"\n"
103     RULE_SNAT = \
104         "-A POSTROUTING {0} {1} {2} {3} {4} -j SNAT --to-source {5} " + \
105           "-m comment --comment \"{6}\"\n"
106     RULE_DNAT = \
107         "-A PREROUTING {0} {1} {2} {3} {4} -j DNAT --to-destination {5} " + \
108           "-m comment --comment \"{6}\"\n"
109
110     SOURCE_HOST = "-s "
111     SOURCE_INTF = "-i "
112     DESTINATION_HOST = "-d "
113     DESTINATION_INTF = "-o "
114     SPORT = "--sport "
115     DPORT = "--dport "
116     PROTOCOL = "-p "
117     IPT_ACCEPT = "ACCEPT"
118     IPT_DROP = "DROP"
119     IPT_REJECT = "REJECT"
120
121     # Variables (list of tuples) used in the switch_elements method. For each
122     # tuple the first element is substituted with the second element
```

```
123     SW_SOURCE = \
124         [(SOURCE_INTF, DESTINATION_INTF), (SOURCE_HOST, DESTINATION_HOST)]
125     SW_DESTINATION = \
126         [(DESTINATION_INTF, SOURCE_INTF), (DESTINATION_HOST, SOURCE_HOST)]
127     SW_SPORT = [(SPORT, DPORT)]
128     SW_DPORT = [(DPORT, SPORT)]
129 
130     ''' Constructor '''
131     def __init__(self, directory):
132         GenericEngine.__init__(self, directory)
133         self.language = "iptables"  # The language is iptables for Netfilter
134         # This dictionary list keeps track of the interface name with the
135         # corresponding net ip
136         self.int_ip = []
137 
138     ''' Ok, let's do the job! '''
139     def translate(self, configuration):
140         # Default rules
141         def_rul = True
142         # Logging
143         logging = True
144         # Established
145         estb = False
146         # This string is used to implement all the bindings between
147         # interfaces and ips
148         bindings = ""
149         # This string is used to implement the filters (with basic rules)
150         filters = self.BASIC_FILTER
151         # This string is used to implement the NAT rules
152         nat = self.BASIC_NAT
153         # When an interface accepts any ip, the ips bound to other interfaces
154         # must be dropped to avoid overlaps.
155         # This string keeps track of the info useful to do such a thing
156         bind_any_drop = self.BIND_ANY_DROP.format("{0}", "127.0.0.0/8", "{1}")
157         # This string is used to set up the mangle rules for the NATs
158         binding_nat = ""
159         # This list keeps track of the interfaces that accept anything
160         intfs = []
161 
162         # Ok, here we have all the conf lines
163         lines = configuration.split('\n')
164         for index, l in enumerate(lines):  # For all the lines
165             if l == "":  # If the line is empty, simply continue
166                 continue
167 
168             parsed = self.parse_line(l)  # Parse the line
169 
170             if parsed[0] == self.OPTN:  # If the line is an OPTN
171                 # We manage the options by setting flags
172                 if parsed[1][0] == "default_rules":  # Default rules
173                     if parsed[1][1] == "yes":
174                         def_rul = True
175                     elif parsed[1][1] == "no":
176                         def_rul = False
177                     else:
178                         print("WARNING: Value for option '%s' not valid: %s"
179                               % (parsed[1][0], parsed[1][1]))
180                 elif parsed[1][0] == "logging":  # Logging
181                     if parsed[1][1] == "yes":
182                         logging = True
183                     elif parsed[1][1] == "no":
184                         logging = False
```

```
185                     else:
186                         print("WARNING: Value for option '%s' not valid: %s"
187                               % (parsed[1][0], parsed[1][1]))
188                 elif parsed[1][0] == "established":  # Established management
189                     if parsed[1][1] == "yes":
190                         estb = True
191                     elif parsed[1][1] == "no":
192                         estb = False
193                     else:
194                         print("WARNING: Value for option '%s' not valid: %s"
195                               % (parsed[1][0], parsed[1][1]))
196                 else:
197                     print("WARNING: Unknown option %s" % parsed[1][0])
198             elif parsed[0] == self.BIND:  # If the line is BIND
199                 # If an interface accepts anything
200                 if parsed[1][1] == "0.0.0.0/0":
201                     intfs.append(parsed[1][0])  # Remember it...
202                 else:  # If it is a "normal" interface
203                     # We set the bound
204                     bindings += \
205                         self.BIND_ACCEPT.format(parsed[1][0], parsed[1][1], l)
206                     # drops set enlarged
207                     bind_any_drop += \
208                         self.BIND_ANY_DROP.format("{0}", parsed[1][1], "{1}")
209                 self.int_ip.append(
210                     {
211                         "int_name": parsed[1][0],
212                         "net_ip": parsed[1][1]
213                     }
214                 )  # Keep track of the int name and ip
215             elif parsed[0] == self.ALLW or \
216                     parsed[0] == self.DROP or \
217                     parsed[0] == self.RJCT or \
218                     parsed[0] == self.TALW:
219                 # If we're dealing with a firewall rule
220                 source = ""  # String for the source
221                 sport = ""  # String for the source port
222                 destination = ""  # String for the destination
223                 dport = ""  # String for the destination port
224                 # String for the protocol, the default case is "-p all"
225                 protocol = self.PROTOCOL + "all"
226                 action = ""  # The action
227                 s_local = False  # Is there a local keyword in the source?
228                 d_local = False  # Is there a local keyword in the destination?
229 
230                 # Let's begin: we get all the details from the rule
231                 rule_detail = self.get_rule_details(parsed[1])
232 
233                 if (rule_detail[0][1] != "" or rule_detail[2][1] != "") \
234                         and not self.migplus:
235                     print("WARNING: MIGNIS+ RULE SPECIFICATION IS A FEATURE STILL IN
                           BETA!!")
236                     print("Check the rules before setting up the firewall.")
237                     self.migplus = True
238 
239                 # If there's an ip in the source, we set "-s <ip>"
240                 if rule_detail[0][0][1] == '-':
241                     source = self.SOURCE_HOST + rule_detail[0][0][2:]
242                 # If there's a local il the source field, we set the flag
243                 elif rule_detail[0][0] == self.LOCAL:
244                     s_local = True
245                 # If there isn't a star in the source field we set "-i <intf>"
```

```
                elif rule_detail[0][0] != self.ANY:
                    source = self.SOURCE_INTF + rule_detail[0][0]

                # If there's an interface (Mignis+) we set also the -i option
                if rule_detail[0][1] != "":
                    source += " " + self.SOURCE_INTF + rule_detail[0][1]

                # Same for destination but with "-d" and "-o" instead of
                # "-s" and "-i" respectively
                if rule_detail[2][0][1] == '-':
                    destination = self.DESTINATION_HOST + rule_detail[2][0][2:]
                elif rule_detail[2][0] == self.LOCAL:
                    d_local = True
                elif rule_detail[2][0] != self.ANY:
                    destination = self.DESTINATION_INTF + rule_detail[2][0]

                # If there's an interface (Mignis+) we set also the -o option
                if rule_detail[2][1] != "":
                    destination += " " + self.DESTINATION_INTF + \
                                   rule_detail[2][1]

                if rule_detail[0][2] != "0":  # Source port
                    sport = self.SPORT + rule_detail[0][2]
                if rule_detail[2][2] != "0":  # Destination port
                    dport = self.DPORT + rule_detail[2][2]

                # If a protocol is specified, we set it with "-p <protocol>"
                if rule_detail[4] != self.ANY:
                    protocol = self.PROTOCOL + rule_detail[4].lower()

                # If the operator is > or <>
                if parsed[0] == self.ALLW or parsed[0] == self.TALW:
                    action = self.IPT_ACCEPT
                elif parsed[0] == self.DROP:  # /
                    action = self.IPT_DROP
                elif parsed[0] == self.RJCT:  # //
                    action = self.IPT_REJECT

                current_rule = ""  # The current rule
                if d_local and not s_local:  # If the destination is "local"
                    current_rule = self.RULE_IN.format(protocol, source, sport,
                                                       dport, rule_detail[5],
                                                       action, l)
                    # <> needs to add a second rule with switched operands
                    if parsed[0] == self.TALW:
                        # "-s" and "-i" becomes "-d" and "-o"
                        source = self.switch_elements(source, self.SW_SOURCE)
                        # "--dport" becomes "--sport"
                        dport = self.switch_elements(dport, self.SW_DPORT)
                        # "--sport" becomes "--dport"
                        sport = self.switch_elements(sport, self.SW_SPORT)
                        current_rule += self.RULE_OUT.format(protocol, dport,
                                                             source, sport,
                                                             rule_detail[5],
                                                             action, l)
                if s_local and not d_local:  # If the source is "local"
                    current_rule = self.RULE_OUT.format(protocol, sport,
                                                        destination, dport,
                                                        rule_detail[5], action,
                                                        l)
                    if parsed[0] == self.TALW:  # <>
                        destination = self.switch_elements(destination,
```

```
                                                    self.SW_DESTINATION)
                dport = self.switch_elements(dport, self.SW_DPORT)
                sport = self.switch_elements(sport, self.SW_SPORT)
                current_rule += \
                    self.RULE_IN.format(protocol, destination, dport,
                                        sport, rule_detail[5], action,
                                        l)
            # Weird case: source and destination are "local"...
            # Here we need 127.0.0.0/8
            if d_local and s_local:
                lip = "127.0.0.0/8"
                ldest = self.DESTINATION_HOST + lip
                lsrc = self.SOURCE_HOST + lip
                current_rule = self.RULE_OUT.format(protocol, sport, ldest,
                                                    dport, rule_detail[5],
                                                    action, l)
                dport = self.switch_elements(dport, self.SW_DPORT)
                sport = self.switch_elements(sport, self.SW_SPORT)
                current_rule += self.RULE_IN.format(protocol, dport,
                                                    lsrc + lip, sport,
                                                    rule_detail[5], action,
                                                    l)
                # <> is really stupid in this case: it doubles the rules
                if parsed[0] == self.TALW:
                    current_rule += current_rule
            if not d_local and not s_local:  # Standard case: no local
                current_rule = \
                    self.RULE_FWD.format(protocol, source, sport,
                                         destination, dport, rule_detail[5],
                                         action, l)
                if parsed[0] == self.TALW:  # <>
                    source = self.switch_elements(source, self.SW_SOURCE)
                    destination = self.switch_elements(destination,
                                                       self.SW_DESTINATION)
                    sport = self.switch_elements(sport, self.SW_SPORT)
                    dport = self.switch_elements(dport, self.SW_DPORT)
                    current_rule += \
                        self.RULE_FWD.format(protocol, destination, dport,
                                             source, sport, rule_detail[5],
                                             action, l)
            filters += current_rule  # The set of filter rules is updated!

            # We consider NATs only when the operator is a >
            if parsed[0] == self.ALLW:
                # If we have a MASQUERADE case
                if rule_detail[1][0] == self.MASQUERADE:
                    if source != "" and source[1] != 's':
                        # if the source is an interface, translate it into
                        # its corresponding net_ip
                        source = \
                            self.SOURCE_HOST + \
                            self.get_ip_by_name(source[3:])[0]["net_ip"]
                    nat += self.RULE_MASQUERADE.format(protocol, source,
                                                       sport, destination,
                                                       dport, l)
                elif rule_detail[1][0] != "":  # A Source NAT is requested
                    if rule_detail[1][0][1] != "-":
                        # A special case: NAT with an interface instead of
                        # a host
                        to_source = "None"
                    else:
                        to_source = rule_detail[1][0][2:]
```

```
370                         if rule_detail[1][2] != "0":  # The sNAT port
371                             to_source += ":" + rule_detail[1][2]
372                         # Again: translate interfaces into its net_ip
373                         if source != "" and source[1] != 's':
374                             source = \
375                                 self.SOURCE_HOST + \
376                                 self.get_ip_by_name(source[3:])[0]["net_ip"]
377                         nat += self.RULE_SNAT.format(protocol, source, sport,
378                                                      destination, dport,
379                                                      to_source, l)
380                     elif rule_detail[3][0] != "":  # Destination NAT
381                         # We avoid to open unnecessary doors...
382                         if destination != "" and destination[1] == 'o':
383                             dest_mangle = \
384                                 self.DESTINATION_HOST + \
385                                 self.get_ip_by_name(
386                                     destination[3:]
387                                 )[0]["net_ip"]
388                         else:
389                             dest_mangle = destination
390                         binding_nat += self.MANGLE_NAT.format(protocol, source,
391                                                               sport,
392                                                               dest_mangle,
393                                                               dport, l)
394                         # Here all the parameters for the NAT rule are set
395                         int_index = destination.find(self.DESTINATION_INTF)
396                         if destination == "" or destination[1] != "d":
397                             to_destination = "None"
398                         else:
399                             if int_index == -1:
400                                 to_destination = destination[3:]
401                             else:
402                                 to_destination = destination[3:int_index - 1]
403                         if dport != "":
404                             to_destination += ":" + dport[8:]
405                         if rule_detail[3][0][1] != '-':
406                             destination = \
407                                 self.DESTINATION_HOST + \
408                                 self.get_ip_by_name(
409                                     rule_detail[3][0]
410                                 )[0]["net_ip"]
411                         else:
412                             if int_index != -1:
413                                 save = " " + \
414                                        self.DESTINATION_INTF + \
415                                        destination[int_index:]
416                             else:
417                                 save = ""
418                             destination = \
419                                 self.DESTINATION_HOST + \
420                                 rule_detail[3][0][2:] + \
421                                 save
422                         if rule_detail[3][2] != 0:
423                             dport = self.DPORT + rule_detail[3][2]
424                         else:
425                             dport = ""
426                         nat += self.RULE_DNAT.format(protocol, source, sport,
427                                                      destination, dport,
428                                                      to_destination, l)
429                 # Here we have the policies
430                 elif parsed[0] == self.PDRP or parsed[0] == self.PRJC:
431                     # Set the action...
```

```
432                 action = self.DROP if parsed[0] == self.PDRP else self.RJCT
433                 # A policy can be translated as a normal firewall rule put below
434                 # all the other rules, so we add a new rule to be evaluated...
435                 # @@@@@@@ IMPORTANT @@@@@@@
436                 # THIS WORKS ONLY IF POLICIES RULES COMES AFTER ALL THE NORMAL
437                 # RULES!
438                 # USE THE MIGNIS COMPILER, DO NOT WRITE RULES BY HAND IN
439                 # INTERMEDIATE REPRESENTATION!
440                 # YOU ARE ADVICED!
441                 new_command = action + ":" + parsed[1][0] + ";" + \
442                               parsed[1][1] + ";" + parsed[1][2] + ";;;0;" + \
443                               parsed[1][3] + ";" + parsed[1][4] + ";" + \
444                               parsed[1][5] + ";;;0;" + parsed[1][6] + ";"
445                 lines.insert(index + 1, new_command)
446             # Custom rules, they are put after the policies... Use with cautions
447             elif parsed[0] == self.CSTM:
448                 filters += parsed[1][0] + "\n"
449 
450         for intf in intfs:  # For all the interfaces that accepts anything
451             comment = "BIND:" + intf + ";0.0.0.0/0"  # Set the comment...
452             # Add the drops and the final accept to the bindings string
453             bindings = bind_any_drop.format(intf, comment) + \
454                        self.BIND_ANY_ACCEPT.format(intf, comment) + \
455                        bindings
456 
457         # Final mangle rules list
458         bindings = self.BASIC_MANGLE + \
459                    (self.DEFAULT_MANGLE if def_rul else "") + \
460                    binding_nat + \
461                    bindings + \
462                    self.MANGLE_LO + \
463                    (self.LOGGING_MANGLE if logging else "")
464         # Final filter rules list - First add established, then all the rest!
465         filters = (filters.format(self.DEFAULT_ESTABLISHED)
466                    if estb else filters.format("{0}"))
467         filters = (filters.format(self.DEFAULT_FILTER)
468                    if def_rul else filters.format("")) + \
469                   (self.LOGGING_FILTER if logging else "")
470 
471         # Return!
472         return filters + "COMMIT\n" + \
473                bindings + "COMMIT\n" + \
474                nat + "COMMIT" + "\n"
475 
476     ''' This method is used to extract the couple {int_name, net_ip} '''
477     def get_ip_by_name(self, name):
478         # List comprehension approach
479         return [item for item in self.int_ip if item['int_name'] == name]
```

Listing B.4: File target_compiler.py

```
#! /usr/bin/env python

__author__ = "Alessio Zennaro"


''' The list of supported target languages, imported as classes '''
from netfilter_engine import NetfilterEngine        # Netfilter/Iptables
from example_engine import ExampleEngine            # Example

import os
import shutil
import sys


''' Main function '''
def main():
    # We must have a parameter stating which target language we want
    if len(sys.argv) == 1 or (len(sys.argv) < 3 and sys.argv[1] != "list"):
        print(
            "Usage: ./target_compiler.py [list | <target_language>] <directory>"
        )
        exit(-1)

    # We save the directory we want to work on and we check that the directory
    # name is well-formed
    main_dir = sys.argv[2] if len(sys.argv) > 2 else ""
    if len(main_dir) > 0 and main_dir[len(main_dir) - 1] != '/':
        print("FATAL: <directory> must end with a '/' character")
        exit(-1)

    if main_dir != "":
        try:
            # If <dir>/final already exists, we delete it.
            # We remove it if it is a file or a dir as well
            dir = main_dir + "/final"
            if os.path.isdir(dir):
                shutil.rmtree(dir)
            elif os.path.isfile(dir):
                os.remove(dir)

            # Ok, a new empty directory is created
            os.makedirs(dir)
        except IOError as _:
            # If something goes wrong, kill everything!
            print("FATAL: I/O error")
            exit(-1)

    # Select the engine
    engine = None
    if sys.argv[1] == "IPTABLES":
        engine = NetfilterEngine(main_dir)
    elif sys.argv[1] == "EXAMPLE":
        engine = ExampleEngine(main_dir)
    # Special value: we obtain a list of supported target languages
    elif sys.argv[1] == "list":
        print("List of supported final target languages:")
        print("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^")
        print("\n")
        print("IPTABLES:\tStandard Netfilter/iptables for Linux OS")
        print(
```

```
            "EXAMPLE:\tAn example used as test that produces a fake final " + \
            "configuration"
        )
        print("\n")
        exit(0)
    else:  # Unknown language
        print("Unknown language. Type './target_compiler list' for the " + \
              "complete list of supported target languages"
        )
        exit(1)

    # If we arrive here, we're done!
    print("\nComplete! Written %d final configurations" % engine.compile())



''' The entry point of the program is the main() function '''
if __name__ == "__main__":
    main()
```

C

SOURCE CODE FOR THE COMPLETE TOOL

Listing C.1: File mignis.py

```
#!/usr/bin/env python

__author__ = 'Alessio Zennaro'

import subprocess
import sys
import os


''' The main function '''
def main():
    if len(sys.argv) < 2:  # This is how the file must be used
        print("Usage: ./mignis.py [list | <language>] <file>")
        exit(0)

    # Show a list of supported language
    if len(sys.argv) == 2 and sys.argv[1] == "list":
        print(
            subprocess.check_output("./tcbin/target_compiler.py list",
                                    shell=True)
            .decode())
        exit(0)
    elif len(sys.argv) == 2 and sys.argv[1] != "list":  # This is a wrong usage
        print("Usage: ./mignis.py [list | <language>] <file>")
        exit(0)

    language = sys.argv[1]  # The language to be used
    file_name = sys.argv[2]  # The complete file name
    # The directory the file is located in
    directory = os.path.dirname(file_name) + "/"
    if directory == "/":  # If there's no directory
        directory = '.' + directory  # Add it as the local one

    # Try to execute the compiler and the translator
    try:
        print(
            subprocess.check_output("./utils/mignis_ic -f " + file_name,
                                    shell=True)
            .decode())
        print(
            subprocess.check_output("./tcbin/target_compiler.py " +
                                    language + " " + directory, shell=True)
            .decode())
    except subprocess.CalledProcessError, e:
        print(e.output)

''' The entry point of the program is the main() function '''
if __name__ == "__main__":
    main()
```

BIBLIOGRAPHY

[1] C. Bozzato, G. Dei Rossi, R. Focardi, and F.L. Luccio. Mignis: A semantic based tool for firewall configuration. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 351 – 365, Vienna, July 2014. IEEE.

[2] P. Adão, R. Focardi, J.D. Guttman, and F.L. Luccio. Localizing firewall security policies. In *2016 IEEE 29th Computer Security Foundations Symposium*, Lisboa, June 2016. IEEE.

[3] A.V. Aho, M.S. Lam, R. Sehti, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education Inc, second edition, 2006.

[4] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, (120):197–213, 1993.

[5] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The ocaml system release 3.1.2. Documentation and user's manual, Institut National de Recherche en Informatique et en Automatique, July 2011.

[6] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[7] S. Calzavara. Compiler construction course website, spring 2016 (http://www.dsi.unive.it/ compilers/wordpress/lectures/), 2016 (Last checked: september 4th 2016).

[8] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The java virtual machine specification - java se 8 edition. Java se 8 release contents, Oracle America, Inc., March 2015.

[9] G. Van Rossum and F.L. Drake. The python language reference - release 3.2.3. Reference manual, Python Software Foundation, June 2012.

GLOSSARY

AST — Acronym that stands for *Abstract Syntax Tree* and it is a tree representation of the syntactic structure of a source code written in a programming language in which each node indicates a specific construct occurring in the original code, 15, 49, 52, 58

DFA — Acronym that stands for *Deterministic Finite Automaton*, a finite-state machine in which for each couple (S_i, a_j) (where S_i is the i-th state and a_j is the j-th symbol) there is only one transition to the next state, 11, 15

dNAT — A particular kind of network address translation in which the destination IP address in a packet is translated into another IP address with the effect of redirecting a connection from one host to another. The source believes to connect to a certain IP while the actual connection is made transparently with another IP, 7, 28, 54

ICMP — Acronym that stands for *Internet Control Message Protocol* and it is a ISO/OSI Layer 3 protocol used by network devices to send error messages, 8

iptables — Program used to define the rules on packets and filters when using the Netfilter framework; sometimes talking about iptables implies the use of the whole Netfilter architecture, 2, 8, 21, 69, 75

lexeme — In compilers theory, sequence of characters that forms a lexical unit; it can be considered as an instance of a specific token as long as the associated pattern is matched, 11, 15, 40

masquerade — A particular kind of network address translation in which multiple private network addresses are able to access the internet using a unique public IP address, translated on the fly, 2, 7, 29, 42, 54, 64

NAT — Acronym that stads for *Network Address Translation* and it consists in modifying the network addresses in the IP protocol datagrams packet header when these packets pass through a routing device, 2, 5, 28, 34, 37, 42, 52, 64, 79

Netfilter — Framework provided by the Linux Kernel that allows for advanced operation like paket filtering or network address translation and through which is possible to create advanced network features like a firewall, 1, 2, 8, 21, 69, 75

OCaml — Objective CAML is an advanced object oriented programming language belonging to the family of ML languages, created in 1996 by Xavier Leroy, Jerome Vouillon e Damien Doligez (among others) as an extension of the CAML language, 5, 46, 66, 70

Python — A high level programming language created by Guido van Rossum in 1991 which is object oriented, strongly and dynamically typed, 70, 78, 83

sNAT — A particular kind of network address translation in which the source IP address in a packet is translated into another IP address before being forwarded to the destination so that this latter sees a source different from the original one. Masquerade can be seen as a particular source NAT, 7, 28, 54

yacc — Acronym that stands for *Yet Another Compiler Compiler* and it is a parser generator that, given a context-free grammar, produces the corrisponding parser written in the C programming language, 17