UNIVERSITY OF ZAGREB
**FACULTY OF ELECTRICAL ENGENEERING AND COMPUTING**

Master Thesis num. 222

# Methods for specification and automatic recognition of network protocols

Dražen Popović

Zagreb, July 2011.

*Umjesto ove stranice umetnite izvornik Vašeg rada.*
*Kako biste uklonili ovu stranicu, obrišite naredbu* `\izvornik`.

*I wish to thank my amazing family for making this dream come true. Thanks to my friends from the 9th for being there and helping me. Special thanks to my mentor Doc.dr.sc Domagoj Jakobović for getting me out of messy situations and for putting up with me.*

So Long, and Thanks for All the Fish! :)

# CONTENTS

**1. Introduction** **1**

**2. Wire definition language** **2**

2.1. Network protocol theory . . . . . 3

2.1.1. Layered architecture . . . . . 5

2.1.2. Network protocol operations . . . . . 6

2.1.3. Network protocol data units . . . . . 7

2.2. Network protocol specification . . . . . 8

2.2.1. PDU specification . . . . . 9

2.3. Abstract and transfer syntax . . . . . 11

2.4. Wire formal definition . . . . . 14

2.5. Wire overview . . . . . 14

2.6. Wire lexical conventions . . . . . 21

2.7. Wire syntax . . . . . 23

2.8. Wire semantics . . . . . 27

2.9. Wire code generation . . . . . 30

**3. Automatic recognition of network protocols** **32**

3.1. Genetic algorithms overview . . . . . 33

3.2. Network protocol genotype . . . . . 34

3.3. Genetic operators . . . . . 35

3.3.1. Evaluation . . . . . 36

3.3.2. Crossover . . . . . 36

3.3.3. Mutation . . . . . 37

**4. Conclusion** **38**

**Bibliography** **41**

**A. Wire lexical definitions** **42**

**B. Wire syntax definitions** **45**

# 1. Introduction

The thesis starts with giving an overview on network protocol theory including protocol design, specification and implementation. This provides the appropriate terminology and a knowledge framework for developing a network protocol specification language, called *Wire*. A custom protocol is developed to demonstrate Wire, thus showing its motivation, purpose and inner workings. Detailed descriptions are provided on making of the Wire compiler including language analysis and code generation. The second part of this thesis analyses the task of automatic recognition of network protocols. The approach taken for solving this problem is evolutionary computation technique, more specifically genetic algorithms. By utilizing the *Evolutionary Computation Framework*, a network protocol genotype and corresponding genetic operators are described and implemented.

# 2. Wire definition language

Wire is a network protocol definition language derived from *Interface Definition Language* (IDL)[1]. It is used to represent an on the wire representation of a certain network protocol in an intuitive and highly abstract manner. The Wire compiler is designed to address the automatic generation of code that handles all of the defined protocol's communications, parsing and construction of packets.

Coding handlers for network protocols is time consuming and highly error prone. One must deal with sanity checks upon packet parsing/construction, integer byte ordering and sizes, various charset encodings, data alignment and padding, error reporting and debugging. Furthermore when considering protocol operations, programmers must take into account memory allocation and buffering, timing, various types of operations such as blocking/nonblocking (synchronous/asynchronous) operations. Programmers take various approaches for tackling coding of network protocol handlers and thus code reusability is low, modularity is weak and uniformity is non-existent (in most cases).

Wire is intended to provide an intuitive way of defining a protocol that can be situated in *Link*, *Network*, *Transport* or *Application* layer. Furthermore the code generated by the compiler fits nicely with the network protocol theory and as such is easy readable. The API provided by the generated code library tends to be simple and easy to use, but of course that depends on the protocol definition. To that point Wire strives to provide an abstraction to its users from underlying networking technologies (*Berkley sockets*, *WinSock*, *TLI*) and host configurations (byte/bit ordering, register sizes, floating point representations, character encodings).

The initial idea for Wire was to create a definition language in which one would define a protocol, run it trough a compiler and get a program library that would handle that particular network protocol. The protocol at hand is already suppose to have a specification such as *Internet protocol* (IP) or even *Hyper Text Transfer Protocol* (HTTP). Thus with a single Wire definition a compiler could generate handlers in mul-

[1] IDL is a specification language used to describe a software component's interface.

tiple languages and/or for various systems and frameworks. For example code generation could be extended to generate *Lua* libraries or more specifically *NMAP NSE* libraries which are written in Lua but exist in a more specialized framework. Also one could generate dissection methods for *Wireshark*. The natural extension to the initial idea is to define your own network protocol for whatever purposes. This makes Wire a definition language and the underlying encoding algorithms a serialization protocol. Similar technologies include *ASN.1*, *JSON*, *XDR*. Wire conceptually differs from mentioned projects in the fact that these projects weren't designed to provide means to define an already existing protocol. One can, by using Wire, specify the on the wire representation of a defined network protocol at hand.

## 2.1. Network protocol theory

What is a protocol? A computer protocol can be defined as a well-defined set of messages (bit patterns or, increasingly today, octet strings) each of which carries a defined meaning (semantics), together with the rules governing when a particular message can be sent. However, a protocol rarely stands alone. Rather, it is commonly part of a *protocol stack*, in which several separate specifications work together to determine the complete message emitted by a sender, with some parts of that message destined for action by intermediate (switching) nodes, and some parts intended for the remote end system. In this *layered* protocol model:

- One specification determines the form and meaning of the outer part of the message, with a *'hole'* in the middle. It provides a *carrier service* (or just *service*) to convey any material that is placed in this 'hole'.
- A second specification defines the contents of the 'hole', perhaps leaving a further 'hole' for another layer of specification, and so on.

2.1 illustrates the TCP/IP stack, where real networks provide the basic carrier mechanism, with the IP protocol carried in the 'hole' they provide, and with IP acting as a carrier for TCP (or the the less well-known User Datagram Protocol - UDP), forming another protocol layer, and with a (typically for TCP/IP) monolithic application layer - a single specification completing the final 'hole'. The precise nature of the service provided by a lower layer (lossy, secure, reliable), and of any parameters controlling that service, needs to be known before the next layer up can make appropriate use of that service. We usually refer to each of these individual specification layers as a *protocol*. Note that in 2.1, the 'hole' provided by the IP carrier can contain either a

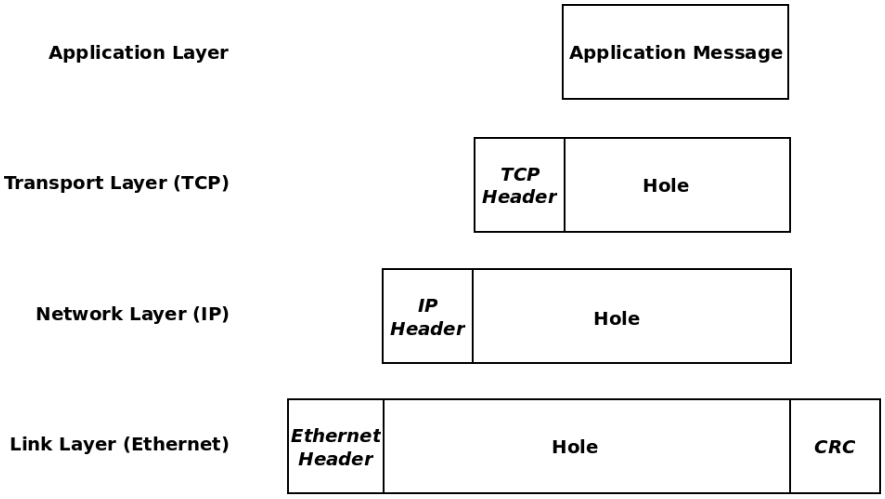


**Figure 2.1:** TCP/IP model ('hole')

TCP message or a UDP message - two very different protocols with different properties (and themselves providing a further carrier service). Thus one of the advantages of layering is in reusability of the carrier service to support a wide range of higher level protocols, many perhaps that were never thought of when the lower layer protocols were developed. When multiple different protocols can occupy a 'hole' in the layer below (or provide carrier services for the layer above), this is frequently illustrated by the layering diagram shown in 2.2

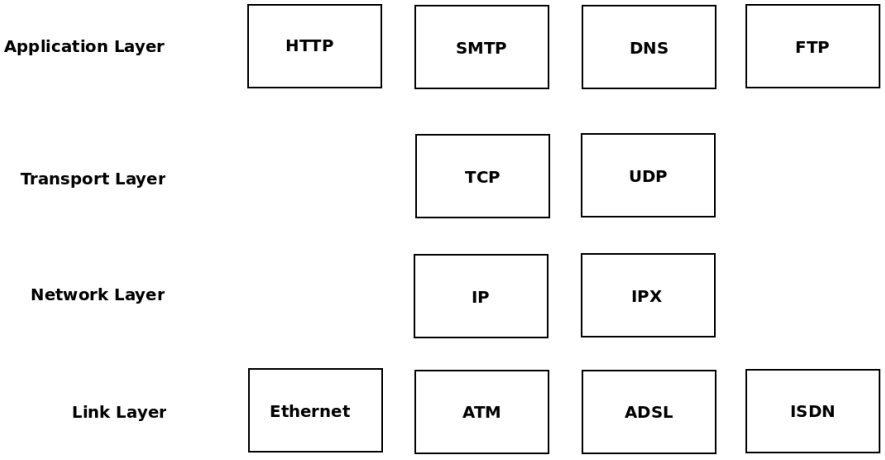


**Figure 2.2:** TCP/IP layering

### 2.1.1. Layered architecture

The layering concept is perhaps most commonly associated with the *International Standards Organization* (ISO) and *International Telecommunications Union* (ITU) architecture or '7-layer model' for *Open Systems Interconnection* (OSI) shown in 2.3. To reduce their design complexity, most networks are organized as a stack of layers or levels, each one built upon the one below it. The number of layers, the name of each layer, the contents of each layer, and the function of each layer differ from network to network. The purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. This is know as *encapsulation*. In a sense, each layer is a kind of 'virtual machine', offering certain services to the layer above it.

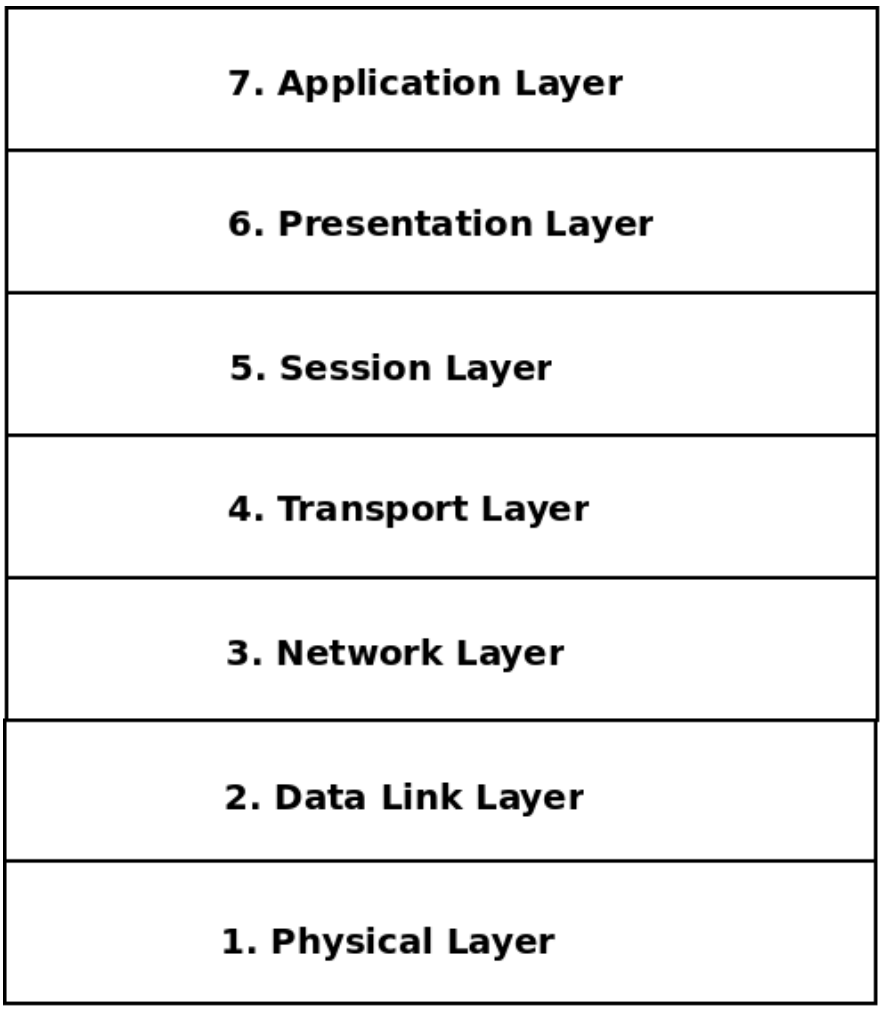


**Figure 2.3:** OSI model (7-layered)

Between each pair of adjacent layers is an interface. The interface defines which

primitive operations and services the lower layer makes available to the upper one. When network designers decide how many layers to include in a network and what each one should do, one of the most important considerations is defining clean interfaces between the layers. Doing so, in turn, requires that each layer perform a specific collection of well-understood functions. In addition to minimizing the amount of information that must be passed between layers, clear-cut interfaces also make it simpler to replace the implementation of one layer with a completely different implementation (eg., all the telephone lines are replaced by satellite channels) because all that is required of the new implementation is that it offer exactly the same set of services to its upstairs neighbor as the old implementation did. In fact, it is common that different hosts use different implementations.

While many of the protocols developed within this framework are not greatly used today, it remains an interesting academic study for approaches to protocol specification. In the original OSI concept in the late 1970s, there would be just 6 layers providing (progressively richer) carrier services, with a final *application layer* where each specification supported a single end-application, with no 'holes'.

### 2.1.2. Network protocol operations

Considering the layered model, lower layer network protocol provides services to a higher layer protocol. This services are exposed trough protocols interface. A protocol then performs certain operations to the point of servicing a higher layer protocol which requested a service. For example TCP provides connection-oriented, ordered and reliable transfer of data from one TCP endpoint to another, for higher level protocol such as *Simple Mail Transfer Protocol* (SMTP) or *File Transfer Protocol* (FTP). This is achieved using operations such as *handshaking*, *acknowledging* and *signaling*.

A simple protocol operation would be to start the operation and then wait for it to complete. But such an approach (called *synchronous* or *blocking* operation) would block the progress of a program while the communication is in progress, leaving system resources idle. The thread of control is blocked within the function performing the protocol operation, and it can use the result immediately after the function returns. This means that the processor can spend almost all of its time idle waiting for a certain protocol operation to complete.

Alternatively, it is possible, to start the operation and then perform processing that does not require that the operation has completed. This type of operation is called *asynchronous* or *non-blocking* operation. Any task that actually depends on the operation

having completed (this includes both using the return values and critical operations that claim to assure that a protocol operation at hand has been completed) still needs to wait for the protocol operation to complete, and thus is still blocked, but other processing which does not have a dependency on the protocol operation can continue. Situations in which a protocol operation should operate in asynchronous mode are those that can get extremely slow, for reasons such as writing or reading from a hard drive (in the context of network file systems).

Additional issue which needs to be addressed concerning protocol operations is the time period in which they must perform. This are called *operation timeouts* and they vary a lot and usually depend on the semantics and the context of the operation itself.

Further we can separate operations to *passive* and *active*, considering if the operation is initiating communication with the other endpoint (eg. active), or is simply waiting for the communication to be initiated by the other endpoint (eg. passive). Also we can utilize terminology such as *server operations* and *client operations*, for passive and active operations, respectively.

### 2.1.3. Network protocol data units

What it boils down to, a protocol operation is actually the exchange of messages. These messages are transmitted across a virtual communication line between endpoints or peers that reside on the same layer and thus 'speak' the same protocol. The abstraction level that the layering approach provides us allows us to think of these peers being directly connected. The message that carries the required semantics among the protocol peers at hand is called the *protocol data unit* (PDU).

A PDU in general consists of a header which contains some kind of protocol-control information and possibly user data of that layer. The other part is considered to be the payload data, formally referred to as *service data unit* (SDU). The semantics and syntax of the SDU is known to the higher layer protocol which is being serviced by the lower layer protocol. The lower layer protocol has no such knowledge and thus SDU is considered as a 'hole' to the protocol at hand.

For example in relation to the OSI model layers, the Physical layer PDU is a *bit*, the Data Link layer PDU is referred to as a *frame*, while the Network layer and the Transport layer use the terms *packet* and *segment*, respectively.

PDUs are commonly *binary-based* or *text-based* (also referred to as *character-based*). Generally with binary-based PDUs protocol gains in speed and bandwidth usage, but in turn has to deal with different integer sizes and sign, floating point rep-

resentations, bit and byte ordering. On the other hand text-based PDUs are relatively simple to handle as they are most commonly ASCII encoded and thus human-readable and easy debugged. Of course it's clear that such PDUs are heavy on bandwidth usage.

## 2.2. Network protocol specification

Protocols can be (and historically have been) specified in many ways. One fundamental distinction is between protocols that utilize character-based PDUs versus binary-based PDUs. Such specifications are commonly referred to as character-based and binary-based specification, respectively:

**Character-based specification** The protocol is defined as a series of lines of ASCII encoded text.

**Binary-based specification** The protocol is defined as a string of octets or of bits.

Character-based protocols are often designed as a *command line* or *statement-based* protocols. The communication of such protocols consist of series of lines of text each of which can be thought of as a command or a statement, with textual parameters (frequently comma separated) within each command or statement. The examples of such text based protocols are HTTP, FTP, POP3, etc.

The common way of defining a text based protocol is with use of *Backus Naur Form* or simply BNF. It is very powerful for defining arbitrary syntactic structures, but it does not in itself determine how variable length items are to be delimited or iteration counts determined. A part of HTTP specification written in BNF is shown in 2.1

**Listing 2.1:** BNF specification of HTTP protocol

```
1  SPACE := ``  ''
2  CRLF := ``\r\n''
3  HTTP–REQUEST := HTTP–REQUEST–LINE HTTP–REQUEST–HEADERS HTTP–MESSAGE–BODY
4  HTTP–REQUEST–LINE := HTTP–METHOD SPACE HTTP–URI SPACE HTTP–VERSION CRLF
5  HTTP–METHOD := ``OPTIONS'' | ``GET'' | ``HEAD'' | ``POST'' | ``PUT'' | ``DELETE'' | ``TRACE'' | ``CONNECT''
6  HTTP–VERSION := ``HTTP/1.0'' | ``HTTP/1.1''
7  HTTP–REQUEST–HEADERS := HTTP–REQUEST–HEADERS HTTP–REQUEST–HEADER | HTTP–REQUEST–HEADER
8  HTTP–URI := ...
9  HTTP–REQUEST–HEADER := ...
10 HTTP–MESSAGE–BODY := ...
```

Binary protocols are more difficult to implement and their wire representation is not human-readable, but generally they are more efficient in both bandwidth usage and speed. For binary-based specification, approaches vary from various *picture-based* methods (2.4) to use of separately defined notation (syntax) with associated application-independent encoding rules (serialization protocols).

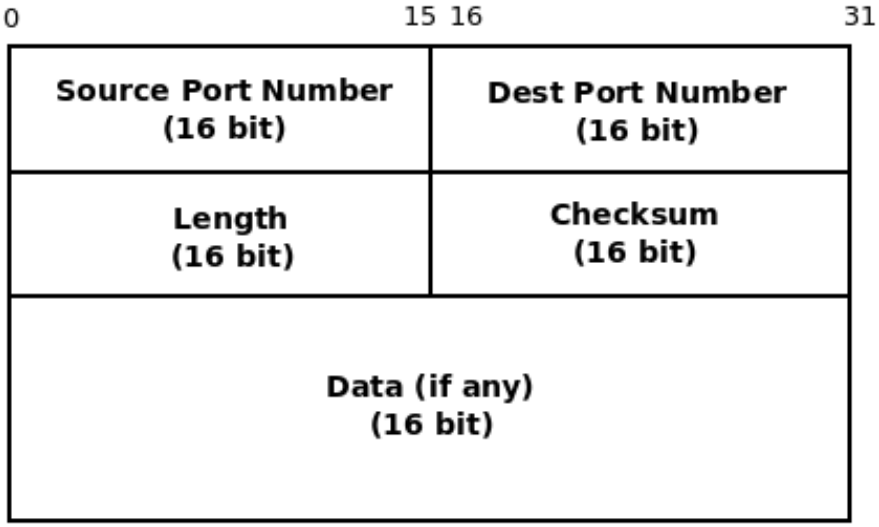


**Figure 2.4:** UDP picture-based specification

The later is called the 'abstract syntax' approach (2.2). This is the approach taken with technologies such as ASN.1, Protocol Buffers, SUN-RPC, ONC-RPC, SOAP etc. It has the advantage that it enables designers to produce specifications without undue concern with the encoding issues, and also permits application-independent tools to be provided to support the easy implementation of protocols specified in this way. Moreover, because application-specific implementation code is independent of encoding code, it makes it easy to migrate to improved encodings as they are developed.

**Listing 2.2:** ASN.1 abstract syntax notation example

```
1  FooProtocol DEFINITIONS ::= BEGIN
2      FooQuestion ::= SEQUENCE {
3          trackingNumber INTEGER,
4          question       IA5String
5      }
6      FooAnswer ::= SEQUENCE {
7          questionNumber INTEGER,
8          answer         BOOLEAN
9      }
10 END
```

### 2.2.1. PDU specification

A PDU is specified using various data types. Let's divide data types into *primitive* types or *basic* types and *constructed* types or *composite* types. Primitive types include integers, floating point numbers, characters, booleans etc. Constructed data types are constructed using primitive data types and other constructed types. They provide enclosure for some data type set. Structures, arrays, strings, unions, enumerators etc., fall into constructed data type category.

Different kinds of computers use different conventions for the ordering of bytes within data types that are multiple of a byte. Some computers put the most significant byte (eg. MSB) within such data type first, this is called *'big endian'* order, and others put it last, thus called *'little endian'* order (eg. LSB). The same goes for bit ordering, all though it's rare to find little endian bit ordering into the wild, both on processor architecture and network protocol specifications. Integer sizes also differ amongst architectures, not to mention floating point representations. Strings have different character encodings (ie. character set or simply charset).

So that machines with different conventions and specifications can communicate, the network protocol specification must clearly define these attributes to every data type transmitted over the network.

Formally we can define integers as a data type which represents some finite subset of mathematical integers (integral data type). When specifying an integer data type one must consider the following attributes:

**Size** Most commonly integer sizes are byte multiples, and as such are usually named with the following size specifiers: *char, short, long, long long* or *hyper* respectively pertaining to sizes of 1,2,4,8 bytes. It's not uncommon to define a bit multiple integer size, for instance 13 bits offset field in the IP PDU specification.

**Byte order** Byte order concept is only valid with byte multiple sized data types so only byte sized integers must have byte ordering defined. Thus called *byte-sized integers*.

**Bit order** Data types that have size defined as bit multiple are called bit-sized data types, therefor such integers are called *bit-sized integers*. Bit order can be specified for both byte-sized and bit-sized integers. When specified for bit-sized integer the bits are arranged accordingly for the integer as a whole. For byte-sized integers the bit order is considered bytewise and thus is set for each byte.

**Sign** An integer can be *signed* or *unsigned*. Signed integers are stored in a computer using *2's complement*. Distinction must be made as integer operations are different for unsigned versus signed integers.

In computer science, floating point describes a system for representing real numbers which support a wide range of values. The following are the attributes applicable to floating point data type:

**Representation** There are several floating point representations used today in computing. Different processor architectures utilize different representations. These are: *IEEE754, VAX, Cray, IBM*...

**Size** The size of a floating point type is usually determined by it's representation, and most commonly are byte sized.

**Byte order** As a byte-sized data type it must have a defined byte order.

**Bit order** Similar to byte-sized integers bit order is defined bytewise (not as a whole).

One more primitive data type is a character. A character data type is used to store symbols such as alphanumeric text, whitespace, punctuation and others. These symbols exist at a higher level of abstraction then integers and floating point numbers. But similarly to these primitive types, characters also must have a mapping from character abstraction to a certain binary representation that can be stored in computer memory or transmitted across a network. Essentially a character is mapped into an integer data type, so we can use object oriented paradigm terminology to describe a character type as being a specialized form of an integer type. As such a character inherits all of the mentioned integer data type attributes to additionally introducing some of its own:

**Character set** Also referred to as a *charset, character encoding, character map* or a *code page*. It represents a mapping of symbols into an integer for the purpose of storing these symbols in the computer memory or transmission over the network. These mapping can be either specified using a predefined set of symbol to number conversion (ASCII) or using an encoding algorithm (Unicode).

## 2.3. Abstract and transfer syntax

The terms *abstract* and *transfer syntax* were primarily developed within the OSI work, and are variously used in other related computer disciplines. These terms will provide us with the terminology for formally defining Wire language and it's purpose.

The following steps are necessary when specifying the messages forming a protocol:

- The determination of the information that needs to be transferred in each message. We here refer to this as the semantics associated with the message.

– The design of some form of data-structure (at about the level of generality of a high-level programming language, and using a defined notation) which is capable of carrying the required semantics. The set of values of this data-structure are called the *abstract syntax* of the messages. We call the notation we use to define this data structure or set of values the *abstract syntax notation*.

– The crafting of a set of rules for encoding messages such that, given any message defined using the abstract syntax notation, the actual bits on the line to carry the semantics of that message are determined by an algorithm specified once and once only (independent of the application). We call such rules *encoding rules*, and we say that the result of applying them to the set of messages for a given application defines a *transfer syntax* for that particular abstract syntax. Therefor, a transfer syntax is the set of bit-patterns to be used to represent the abstract values in the abstract syntax, with each bit-pattern representing just one *abstract value*.

So to simplify a little bit, let's say, for example, that we wish to make a notation to declare an integer type and assign it a value. To that point let us borrow the notation that C language uses or simply:

**Listing 2.3:** C abstract syntax notation

```
1 int a = 1;
```

We can now say that the value '1' is an abstract value that represents an integer value. The set of these abstract values (ie. ...-1, 0, 1, 2, 3, 4, 5, 6, 7...) is called the abstract syntax. The notation used to declare and define an instance of an abstract syntax is called an abstract syntax notation.

The usage of term 'abstract' is totally justified, considering we are dealing with the abstraction of integers, floating point numbers, characters etc. The representation used to store an abstract syntax in computer memory is called the *concrete syntax*. For example IEEE754 floating point representation is one of the concrete syntaxes for storing floating point numbers.

Abstract syntaxes should be independent of the concrete syntaxes which can and usually do differ amongst different machines.

The transfer syntax is the representation used to transfer the abstract syntax over the communication line. A certain instance of the abstract syntax or the abstract value must have a unique transfer value so it can be restored on the other endpoint. The transfer syntax must take in order the differences in concrete syntaxes between communicating peers, therefore the transfer syntax must correspond to some sort of protocol. The

protocol or algorithm or encoder or whatever you might call it, which maps the abstract syntax to a corresponding transfer syntax or even a concrete syntax, which carries its semantics, is called *serialization*.

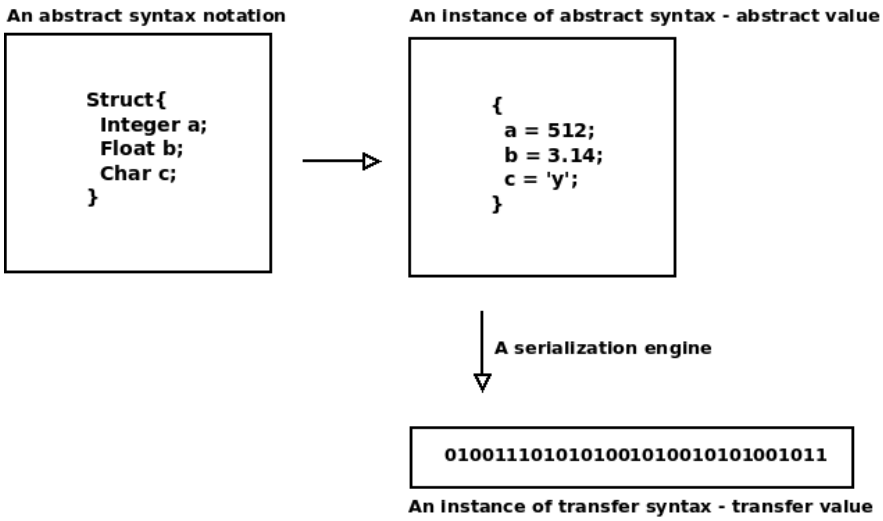


**Figure 2.5:** Syntax relations

The figure 2.5 illustrates the mentioned concepts and their relations.

What's left is to list some technologies and formally describe them using these newly learned terminology. First lets mention the *Abstract Syntax Notation One* technology (ASN.1), which is, as you may noticed, named quite literal. Some of the abstract data types that the ASN.1 provides are:

1. Basic (primitive) types *boolean, integer, real, enumerated, bit string, octet string, null* ...

2. Constructed types include *sequence, set, choice*...

The ASN.1 is an abstract syntax notation which uses several different serialization protocols to produce the transfer syntax:

1. Basic Encoding Rules (BER)

2. Canonical Encoding Rules (CER)

3. Distinguished Encoding Rules (DER)

4. XML Encoding Rules (XER)

5. Packed Encoding Rules (PER)

6. Generic String Encoding Rules (GSER)

Other popular technology that is utilized by the SUN-RPC remote procedure call protocol is the *External Data Representation* (XDR). XDR also includes an abstract syntax notation that defines basic data types such as *integer and hyper, float and double, quadruple, bool* and constructed data types such as *structures, enumerations and unions*. The serialization for each data type is specified in RFC4506.

*JavaScript Object Notation* or simply JSON is widely known and popular data interchange technology used in Web today. It consists of an abstract syntax notation that is a subset of JavaScript scripting language. All though its main purpose is the serialization and transmission of JavaScript objects between a server and a web application (client), JSON is language independent. The serialized objects,aka the transfer syntax, is in human-readable text form.

## 2.4. Wire formal definition

By using the terminology and concepts elaborated in the previous sections, we will formally define the Wire project. First of all Wire is a computer language, more precisely a subset which is called a *definition language*. It's a *domain-specific* or a *special-purpose* language (we'll get to that later on), oppose to general-purpose languages such as C or Java. Also, to specify a bit deeper, Wire provides an abstract syntax notation.

The idea behind Wire is to provide a language which is used to describe/define (ie. definition language) the *on the wire representation* of an arbitrary network protocol. One could say that Wire is used to define the transfer syntax for some protocol. So an abstract syntax notation for describing a transfer syntax.

Wire language compiler takes a specification of a certain protocol written in Wire and produces code that handles the defined protocol. By that I mean generates code to easily build and dissect protocol data units as well as handle all of the defined protocol operations.

## 2.5. Wire overview

For the sake of demonstrating Wire, we will develop a custom network protocol. The development process and the making of the final specification will help us explain the purpose and the inner workings of Wire. Also it will show us some general guidelines and steps needed when designing a protocol.

So let's call this new network protocol *'Math'*, it will resemble a remote procedure call protocol which will offer a mathematical service. So we've already defined:

1. Name - 'Math'

2. Service - Mathematical operations.

So let's start writing a Wire definition of our 'Math' protocol. This is the simplest protocol definition in Wire:

**Listing 2.4:** Wire example - Designing 'Math' (step 1)

```
1 [
2   //protocol attributes
3 ] protocol Math{
4   //data type definitions
5   //operation declarations
6 }
```

This represents nothing yet, but be patient...we'll get there. First we can notice line comments similar to those found in C syntax. Second thing to notice is the *'protocol'* keyword which is used to define a new protocol named 'Math'. What precedes this keyword is a pair of square brackets. These will hold a list of attributes that are applicable to a protocol definition. Actually every Wire component can have attributes applied to it, which attributes are applicable where we'll learn gradually.

Let's define our service a bit more. So we wish to provide basic mathematical operations such as addition, subtraction, multiplication, division and power:

**Listing 2.5:** Wire example - Designing 'Math' (step 2)

```
1 [
2   //protocol attributes
3 ] protocol Math{
4   //data type definitions
5   //operation declarations
6   operation Add();
7   operation Sub();
8   operation Mul();
9   operation Div();
10  operation Pow();
11 }
```

The *'operation'* keyword is assigned the honor of declaring our protocol operations. Currently these operations are dumb as they have an empty arguments list. The operation declaration can only have arguments of *'pdu'* data type. This is reasonable as an operation is the exchange of messages between peers, these messages are called protocol data units and the 'pdu' data type embodies this concept in Wire.

So a pdu definition is defined using the *'pdu'* keyword. I decided to define one general PDU that can capture all of the required semantics. I called it 'Math' protocol data unit. One could decide to go with, for example, two PDUs, one which will carry

the request information and the other that will hold the response information. Notice that this is a design issue and as such falls under personal preference.

**Listing 2.6:** Wire example - Designing 'Math' (step 3)

```
1  [
2    //protocol attributes
3  ] protocol Math{
4    //data type definitions
5    pdu Math{};
6    //operation declarations
7    operation Add([push] pdu Math math_req, [pull] pdu Math math_rep);
8    operation Sub([push] pdu Math math_req, [pull] pdu Math math_rep);
9    operation Mul([push] pdu Math math_req, [pull] pdu Math math_rep);
10   operation Div([push] pdu Math math_req, [pull] pdu Math math_rep);
11   operation Pow([push] pdu Math math_req, [pull] pdu Math math_rep);
12 }
```

Now our operations have a valid argument list. Each pdu local declaration in the argument list has been applied an attribute. These *'push'* and *'pull* attributes are only applicable to pdu declarations that are placed in the argument list of an operation declaration:

**push** It's a notion used to define that a pdu is being sent to the carrier service which conveys it to the other endpoint. The carrier service is situated at a lower layer, so the PDU is virtually being *'pushed'* down.

**pull** Similar to 'push', with the exception that the pdu is being received from the other endpoint, or *'pulled'* up from the lower layer protocol.

Each operation pushes one pdu, similar to function arguments, and pulls one, ie. function return value. Obviously these pdus must be designed so that are able to carry all the required information such as integer number and/or real number arguments, result of the mathematical operation and quite possibly some sort of error messages that indicate an exception (for example division by zero).

**Listing 2.7:** Wire example - Designing 'Math' (step 4)

```
1  [
2    //protocol attributes
3  ] protocol Math{
4
5    //data type definitions
6    enum PDUType{
7      REQUEST = 0,
8      REPLY = 1
9    };
10
11   struct MathReq {};
12
13   struct MathRep{};
14
15   pdu Math{
16     enum PDUType epdu_type;
17     union <epdu_type> {
18       case REQUEST:
```

```
19          struct MathReq smathreq;
20        case REPLY:
21          struct MathRep smathrep;
22        default:
23          exception("epdu_type: value not used");
24      };
25    };
26
27    //operation declarations
28    operation Add([push] pdu Math math_req, [pull] pdu Math math_rep);
29    operation Sub([push] pdu Math math_req, [pull] pdu Math math_rep);
30    operation Mul([push] pdu Math math_req, [pull] pdu Math math_rep);
31    operation Div([push] pdu Math math_req, [pull] pdu Math math_rep);
32    operation Pow([push] pdu Math math_req, [pull] pdu Math math_rep);
33  };
```

We extended our *'Math'* pdu with the *union* construct which allows us to define conditional structuring. A union declaration has a *switch* which determines the proper structure at runtime. The switch value is used to check the equality of *case*s so the code can unambiguously process the structure. This particular union declaration instance is an example of the, so called, *anonymous* union declaration, oppose to a *named* union declaration.

Another handy data type we introduced with this progress is the enumeration data type. It's no novelty, but it's usage results in more elegant definitions. The enum definition takes a list of names (ie. identifiers) for integer constants who will later on be referenced by that name.

Our *'Math'* pdu definition is now equipped with sufficient information to carry both the request and response information. We've reached the second checkpoint in designing a network protocol:

1. Interface – We've designed the exact interface to our mathematical service. It consists of operations: *Add, Sub, Mul, Div, Pow*.

2. PDU – Decided on the structure of the protocol data units. Preferred on a single PDU definition that can carry a more general information.

What's next is to exactly define the information that will be carried within request and reply structures. So we introduce the structure definition notation with the *'struct'* keyword. Syntactically it's no different from the pdu definition. Another primitive type used is the string defined using the *'string'* keyword. We used it to carry the error message.

**Listing 2.8:** Wire example - Designing 'Math' (step 5)

```
1  [
2    //protocol attributes
3  ] protocol Math{
4
5    //data type definitions
```

```
6   enum PDUType{
7     REQUEST = 0,
8     REPLY = 1
9   };
10
11  enum ReplyType{
12    FAILURE,
13    SUCCESS
14  };
15
16  enum NumberType{
17    INTEGER,
18    REAL
19  };
20
21  struct MathReq {
22    enum NumberType enumber_type;
23    unsigned int narguments;
24    union <enumber_type> {
25      case INTEGER:
26        unsigned int size_arg;
27        signed int sint_args[narguments];
28      case FLOAT:
29        unsigned int size_arg;
30        float fp_args[narguments] ;
31      default:
32        exception("enumber_type: value not used");
33    };
34  };
35
36  struct MathRep{
37    enum ReplyType ereply_type;
38    union <ereply_type>{
39      case FAILURE:
40        string   strerror;
41      case SUCCESS:
42        enum NumberType enumber_type;
43        union <enumber_type> {
44          case INTEGER:
45            unsigned int size_res;
46            signed int sint_res;
47          case REAL:
48            unsigned int size_res;
49            float fp_args
50          default:
51            exception("enumber_type: value not used");
52      };
53    default:
54      exception("enumber_type: value not used");
55    };
56  };
57
58  pdu Math{
59    enum PDUType epdu_type;
60    union <epdu_type> {
61      case REQUEST:
62        struct MathReq smathreq;
63      case REPLY:
64        struct MathRep smathrep;
65      default:
66        exception("epdu_type: value not used");
67    };
68  };
69
70  //operation declarations
71  operation Add([push] pdu Math math_req, [pull] pdu Math math_rep);
72  operation Sub([push] pdu Math math_req, [pull] pdu Math math_rep);
73  operation Mul([push] pdu Math math_req, [pull] pdu Math math_rep);
74  operation Div([push] pdu Math math_req, [pull] pdu Math math_rep);
75  operation Pow([push] pdu Math math_req, [pull] pdu Math math_rep);
76 };
```

Notice the *exception* statement. It's a Wire construct used to designate an exception state of some kind. This *function call* like statement takes variable length list of arguments that are passed to the runtime that will eventually handle the exception at hand. What's considered an exception is decided by the protocol designer. I used it to designate the occurrence of invalid value.

At this point we've, semantically, fully defined the protocol from an abstraction level that only deals with information and its exchange trough operations. Needless to say we've defined the abstract syntax of the Math protocol. In the process we've encountered every Wire component: *protocol definition, operation declarations and data type definitions*. Primitive data types: *integers, floats and strings* and constructed data types: *protocol data units, structures, unions, arrays and enumerations*.

Now it's time to define and utilize the Wire attribute concept to exactly define the on the wire representation of the protocol at hand. Attributes can be applied to every Wire component. They are used to semantically link scope related objects, define sanity checks and to give instructions to both the Wire serialization engine and the communication engine. One could say that by using attributes you can exactly define the transfer syntax of a protocol.

First let's talk to the communication engine. What we need to define for Math protocol is the carrier service. The carrier service is a lower layer protocol which is assigned the job of carrying the Math PDUs to the other Math endpoint. To decide on the carrier service for Math, I took into consideration the following:

1. Math endpoints must be able to communicate over the IP networks.

2. The information exchanged over the wire is sensitive in a way that it must be correctly transported to the other side, thus we want reliable and ordered transport service.

So a common practice when met with such requirements is to choose the *Transmission Control Protocol*. TCP resides on top of the IP protocol and uses the *port* addressing concept to deliver data from one process to another. So let's choose our port number to be 31337 (elitenzi).

Wire uses the *endpoint* attribute for defining protocol endpoint information. Generally endpoint attribute takes a name and the addressing information of the carrier protocol. Note that a protocol can specify any number of carrier services, but practically it depends on the communication engine and what services it supports.

**Listing 2.9:** Wire example - Designing 'Math' (step 6)

```
1  [
2    //protocol attributes
3    endpoint(''tcp:31337''),
4    size(4),
5    byte_order(''MSB''),
6    bit_order(''MSB'')
7  ] protocol Math{
8
9    //data type definitions
10   [size(1)] enum PDUType{
11     REQUEST = 0,
12     REPLY = 1
13   };
14
15   enum ReplyType{
16     FAILURE,
17     SUCCESS
18   };
19
20  enum NumberType{
21     INTEGER,
22     REAL
23   };
24
25   struct MathReq {
26     enum NumberType enumber_type;
27     [size(2), byte_order(''LSB'')] unsigned int narguments;
28     union <enumber_type> {
29       case INTEGER:
30         [range(1,32)] unsigned int size_arg;
31         [size_bits(size_arg)] signed int sint_args[narguments];
32       case FLOAT:
33         [list(32,64)] unsigned int size_arg;
34         [size_bits(size_arg), fp_rep(''IEEE754'')] float fp_args[narguments] ;
35       default:
36         exception("enumber_type: value not used");
37     };
38   };
39
40   struct MathRep{
41     enum ReplyType ereply_type;
42     union <ereply_type>{
43       case FAILURE:
44         [charset(''ASCII''), delimiter(''\0'')] string   strerror;
45       case SUCCESS:
46         enum NumberType enumber_type;
47         union <enumber_type> {
48           case INTEGER:
49             [range(1,32)] unsigned int size_res;
50             [size_bits(size_res)] signed int sint_res;
51           case REAL:
52             [list(32,64)] unsigned int size_res;
53             [size_bits(size_res), fp_rep(''IEEE754'')] float fp_args
54           default:
55             exception("enumber_type: value not used");
56       };
57     default:
58       exception("enumber_type: value not used");
59     };
60   };
61
62   pdu Math{
63     enum PDUType epdu_type;
64     union <epdu_type> {
65       case REQUEST:
66         struct MathReq smathreq;
67       case REPLY:
68         struct MathRep smathrep;
69       default:
70         exception("epdu_type: value not used");
71     };
72   };
73
```

```
74    //operation declarations
75    [timeout(5)] operation Add([push] pdu Math math_req, [pull] pdu Math math_rep);
76    [timeout(5)] operation Sub([push] pdu Math math_req, [pull] pdu Math math_rep);
77    [timeout(5)] operation Mul([push] pdu Math math_req, [pull] pdu Math math_rep);
78    [timeout(5)] operation Div([push] pdu Math math_req, [pull] pdu Math math_rep);
79    [timeout(5)] operation Pow([push] pdu Math math_req, [pull] pdu Math math_rep);
80 };
```

The timeout attribute takes the number of seconds as the sole argument, and is applicable to operation objects. This states the maximum amount of time that the operation has to finish, timing from the invocation moment. If the operations has failed to do so for whatever reason the timeout exception is raised and handled by the application logic.

The *attributes applicability* is defined as the context in which the attribute is valid, or equally as the list of objects that are directly influenced by the attribute. For example the floating point representation attribute (*'fp_rep'*) is applicable only to float local declarations, oppose to integer local declarations. An attribute can be defined as a *general attribute*, which is useful for general application. So for example if we define byte order attribute (*'byte_order'*) in the protocol definition attribute list it means that every enclosed object receives this attribute (except when overriden by a more specific attribute statement).

We've defined a few general attributes for the Math protocol. We've set the general byte order and bit order to big endian, while defining the default primitive size to 4 bytes.

We've listed a few of the command attributes that are used to define the transfer syntax of our protocol, by commanding the serialization and the communication engine. There are also attributes that define certain semantic checks that must be performed for a given object at runtime. Range and list check for integer and float declarations are example of such attributes.

## 2.6. Wire lexical conventions

Lexical analysis is the process of converting a sequence of characters into a sequence of lexical units called *tokens*. Wire uses *GNU Flex* tool to generate the Wire tokenizer. Flex is really convenient for processing textual files as it allows users to simply describe tokens using regular expressions. Appendix A.1 holds the flex source file which lists all of the defined Wire tokens and their corresponding regular expressions.

Let's introduce some, more relevant, lexical conventions for Wire users. The following lists the reserved words:

**Listing 2.10:** Wire keywords

```
byte     enum     operation
uint     struct   import
sint     union    typedef
float    pdu      default
string   protocol
```

A Wire identifier follows C syntax rules, so the following are valid examples of Wire identifiers:

**Listing 2.11:** Wire identifier

```
ident1234
ident_1234
ident1234_
_ident1234
```

A valid character set for a Wire identifier consists of alphanumeric Al characters and the underscore. Note that the first character must be a letter or an underscore sign.

The last thing that's left are the numeric constants and the string constants. Integer constants can be stated in decimal, hexadecimal, binary and octal form:

**Listing 2.12:** Wire integer constants

```
255
0xFF or 0xff
0377
0b11111111
```

Floating point number constants look like:

**Listing 2.13:** Wire floating point constants

```
3.14
3.14e10
3.14E10
3.14e-10
```

The 'e' or 'E' notation is used for defining an exponent. A string constant is enclosed between the two double apostrophe signs:

**Listing 2.14:** Wire string constants

```
''This is a string constant''
''Wire uses the \\ as the escape symbol''
```

Wire string constants follow the same rules of C strings.

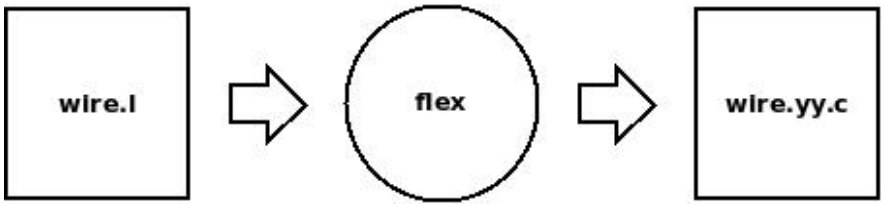


**Figure 2.6:** Wire tokenizer compilation process

2.6 shows the compilation process which generates the Wire tokenizer code. The Flex tool takes a *'wire.l'* file as the input. This file holds token descriptions written in Flex syntax. Flex processes this file and produces the actual C code that does the lexical analysis, *'wire.yy.c'*. The output file contains *'yylex()'* function which upon invocation returns the next token in the assigned stream.

## 2.7. Wire syntax

Parsing or formally syntax analysis is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Wire utilizes the *GNU Bison* tool which reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser in C which reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison generates LALR parsers.

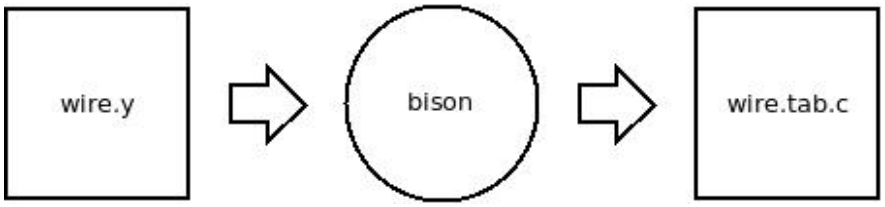


**Figure 2.7:** Wire parser compilation process

Figure 2.7 demonstrates the translation process which generates the Wire parser code. Bison input is a *'wire.y'* file which contains the syntax definition written in Bison variance of BNF syntax definition notation. File *'wire.tab.c'* contains the C code that implements the parsing logic for our language.

My intention was to make Wire syntax simple and intuitive, clean and easy memorable. Appendix B.1 contains a numbered list of all Wire grammar rules.

This chapter holds a brief explanation for every syntactical grouping found in Wire syntax.

The top level grouping is a protocol grouping, and is defined as follows:

**Listing 2.15:** Wire protocol definition

```
3  protocol: attribute_list_opt tPROTOCOL tIDENTIFIER '{' protocol_body_opt '}'
4  protocol_body_opt: protocol_body
5                   | /* empty */
6  protocol_body: protocol_body_component ';'
7               | protocol_body protocol_body_component
8  protocol_body_component: type_definition
9                         | operation_declarator
```

The *'tPROTOCOL'* token represents the *'protocol'* keyword and *'tIDENTIFIER'* token the identifier lexical unit. The protocol body is constructed of components that are separated by a semicolon (';'). These components can be a type definition or a operation declaration. Notice that the protocol definition is prepended an optional attribute syntactical grouping:

**Listing 2.16:** Wire attribute list

```
10 attribute_list_opt: '[' attribute_list ']'
11                   | /* empty */
12 attribute_list: attribute
13               | attribute_list ',' attribute
14 attribute: tIDENTIFIER
15          | tIDENTIFIER '(' attribute_argument_list ')'
16 attribute_argument_list: attribute_argument
17                        | attribute_argument_list ',' attribute_argument
18 attribute_argument: const_exp
```

The attribute list construct is surrounded by the square brackets. An attribute can be specified in one of two forms: with arguments or without arguments. When specified with arguments, these arguments are comma separated and must resolve to syntactical grouping that represents the constant expression.

**Listing 2.17:** Wire type definition

```
19 type_definition: enum_definition
20                | union_definition
21                | struct_definition
22                | pdu_definition
```

The type definition groups rules for definitions of enumerator, union, structure and protocol data unit constructs. The enumerator definition looks like:

**Listing 2.18:** Wire enumerator definition

```
23 enum_definition: attribute_list_opt tENUM tIDENTIFIER '{' enum_body '}'
24 enum_body: enum_body_component
25          | enum_body ',' enum_body_component
26 enum_body_component: tIDENTIFIER '=' const_exp
27                    | tIDENTIFIER
```

As expected the optional attribute list precedes the enumerator definition. The *'tENUM'* token represents the *'enum'* keyword. Enumerator body components are

comma separated and can be specified in one of two forms: with or without explicit value assignment. If no explicit assignment is specified, for a component, the numbering or indexing is calculated considering the offset from last of such assignment (or from zero if no assignment is specified).

**Listing 2.19:** Wire union definition

```
1   28 union_definition: attribute_list_opt tUNION tIDENTIFIER '{' union_body '}'
2   29 union_body: union_body_component
3   30           | union_body union_body_component
4   31 union_body_component: const_exp ':' local_declarator_list
5   32                     | tDEFAULT ':' local_declarator_list
```

The union component is a defined as a case component. The case is a constant expression which is followed by a group of local declarations. The union switch is defined on union declaration. Its purpose is to provide a notation for conditional processing in Wire by checking the equivalence with the listed case constant expressions.

A structure body consists of local declarations separated by a semicolon:

**Listing 2.20:** Wire structure definition

```
1   33 struct_definition: attribute_list_opt tSTRUCT tIDENTIFIER '{' struct_body '}'
2   34 struct_body: struct_body_component
3   35            | struct_body struct_body_component
4   36 struct_body_component: local_declarator ';'
```

Syntactically there is no difference between a structure definition and a protocol data unit definition:

**Listing 2.21:** Wire pdu definition

```
1   37 pdu_definition: attribute_list_opt tPDU tIDENTIFIER '{' pdu_body '}'
2   38 pdu_body: pdu_body_component
3   39         | pdu_body pdu_body_component
4   40 pdu_body_component: local_declarator ';'
```

Next lets look at the local declarator construct:

**Listing 2.22:** Wire local declarators

```
1   41 local_declarator: primitive_local_declarator
2   42                 | constructed_local_declarator
3   43                 | anon_local_declarator
4   44 local_declarator_list: local_declarator ';'
5   45                      | local_declarator_list local_declarator ';'
6   46 primitive_local_declarator: attribute_list_opt tBYTE tIDENTIFIER array_declarator_opt
7   47                           | attribute_list_opt tFLOAT tIDENTIFIER array_declarator_opt
8   48                           | attribute_list_opt tSTRING tIDENTIFIER array_declarator_opt
9   49                           | attribute_list_opt tUINT tIDENTIFIER array_declarator_opt
10  50                           | attribute_list_opt tSINT tIDENTIFIER array_declarator_opt
11  51 constructed_local_declarator: attribute_list_opt tENUM tIDENTIFIER tIDENTIFIER array_declarator_opt
12  52                             | attribute_list_opt tSTRUCT tIDENTIFIER tIDENTIFIER array_declarator_opt
13  53                             | attribute_list_opt tUNION tIDENTIFIER tIDENTIFIER array_declarator_opt
14  54                             | attribute_list_opt tPDU tIDENTIFIER tIDENTIFIER array_declarator_opt
15  55 anon_local_declarator: attribute_list_opt tUNION '<' const_exp '>' '{' union_body '}'
16  56 array_declarator_opt: '[' const_exp ']'
17  57                     | /* empty */
```

A local declaration includes a primitive, constructed and anonymous local declarator. The term *'local'* is used to denote the scope of declared objects. A primitive local declaration is simply a declaration of object that is of primitive type (such as string or integer). Similarly the constructed declaration is a declaration of an object that is of some constructed data type (for example structure). The anonymous local declaration, oppose to a named declaration, is a handy syntactical construct used to declare an union local declaration without a *'name'*.

All of the declarations can be appended an array declarator which takes a constant expression to denote the size of the array. Finally, what does the constant expression look like:

**Listing 2.23:** Wire local declarators

```
1   62 const_exp: integer_const_exp
2   63          | float_const_exp
3   64          | string_const_exp
4   65          | identifier
5   66          | arithmetic_exp
6   67          | relational_exp
7   68          | logical_exp
8   69          | bitwise_exp
9   70 float_const_exp: tFLOATCONST
10  71 string_const_exp: tSTRINGCONST
11  72 integer_const_exp: tINTCONST
12  73 arithmetic_exp: const_exp '+' const_exp
13  74               | const_exp '-' const_exp
14  75               | const_exp '*' const_exp
15  76               | const_exp '/' const_exp
16  77               | const_exp '%' const_exp
17  78 relational_exp: const_exp '>' const_exp
18  79               | const_exp '<' const_exp
19  80               | const_exp tRELEQU const_exp
20  81               | const_exp tRELNEQU const_exp
21  82               | const_exp tRELGE const_exp
22  83               | const_exp tRELLE const_exp
23  84 logical_exp: '!' const_exp
24  85            | const_exp tLOGAND const_exp
25  86            | const_exp tLOGOR const_exp
26  87 bitwise_exp: '~' const_exp
27  88            | const_exp '&' const_exp
28  89            | const_exp '|' const_exp
29  90            | const_exp '^' const_exp
30  91            | const_exp tBITSR const_exp
31  92            | const_exp tBITSL const_exp
32  93 identifier: tIDENTIFIER
33  94           | identifier '.' tIDENTIFIER
```

As we can see a constant expression expands to several expressions. So we have constant expression for primitive types, such as integers, floating point numbers and strings. There is an identifier expression which must resolve to certain object declared in a related scope. Furthermore Wire provides arithmetic, relational, logical and bitwise expressions.

**Listing 2.24:** Wire operation declarator

```
1   58 operation_declarator: attribute_list_opt tOPERATION tIDENTIFIER '(' operation_arg_list ')'
2   59 operation_arg_list: operation_arg ','
3   60                   | operation_arg_list operation_arg
```

The operation declaration construct, similar to function declaration construct found in other languages, receives an comma separated argument list. Syntax enforces that argument list of an operation contains only pdu object local declarations.

After a successful reduction of our input Wire definition to the starting parsing symbol an abstract syntax tree is created for that particular Wire definition instance. This abstract syntax tree is passed over to the next step of the language analysis, the semantic check.

## 2.8. Wire semantics

The semantic check stage of Wire language analysis consists of several logically divided phases:

- Attribute check – involves attribute arguments check and attribute applicability check.
- Local declaration scope check.
- Data type definition and operation declaration name check.
- Constant expression check.

Of course before any of that is possible, we must first formally define the Wire attribute concept and the related terminology:

**Definition 1.** *Attribute is a Wire construct or a notation used to provide semantic extensions the language components and to specify instructions to both the Wire serialization engine and communication engine.*

Attribute is said to be applied to a certain Wire component:

**Definition 2.** *Attribute applicability is defined as the context in which the attribute is valid, or equally as the list of Wire components that are directly influenced by the attribute.*

For example the floating point representation attribute is applicable only to floating point number declarations, oppose to, for example, integer declarations.

I decided to introduce yet another attribute concept, which comes in quite handy:

**Definition 3.** *A general attribute is an attribute that can be placed in the attribute list of a Wire component that has no applicability relation to that attribute.*

This allows us to define such attribute in a more general context. So for example if we define byte order attribute in the protocol definition attribute list it means that every enclosed component, implicitly, receives this attribute (except when overriden by a more specific attribute definition).

**byte_order(string)** General serialization attribute applicable to uint,sint,float,string local declarations.

**bit_order(string)** General serialization attribute applicable to uint,sint,float,string local declarations.

**fp_rep(string)** General serialization attribute applicable to float local declarations.

**char_enc(string)** General serialization attribute applicable to string local declarations.

**size(uint)** General serialization attribute applicable to uint,sint,float,string local declarations.

**size_bits(uint)** General serialization attribute applicable to uint,sint,float,string local declarations.

**align(uint)** Non-general serialization attribute applicable to any local declaration and any type definition.

**delimiter(any)** Non-general serialization attribute applicable to array and string declarations.

**endpoint(string)** Non-general communication attribute applicable to protocol definition.

**timeout(uint)** General communication attribute applicable to operation declaration.

**exception(any, (any)...)** Non-general sanity attribute applicable to any local declaration.

**list((any)...)** Non-general sanity attribute applicable to any local declaration.

**range(any, any)** Non-general sanity attribute applicable to any sint,uint,float local declarations.

**const(any)** Non-general sanity attribute applicable to any local declaration.

**md5(any)** Non-general value attribute applicable to any uint local declaration.

Lets move on to describe the Wire *local declaration scoping*. Lets first define the term scope in the broad context of computer programming:

**Definition 4.** *Scope is an enclosing context where values and expressions are associated.*

Typically, scope is used to define the extent of information hiding, that is, the visibility or accessibility of variables from different parts of the program. Scopes can contain declarations or definitions of identifiers, statements or expressions, nest or be nested. In the context of Wire, a scope is as simple as:

**Definition 5.** *Wire scope contains local declarations of a type definition or a operation declaration.*

So every local declaration within a Wire component that holds them, is assigned a scope, thus making the local declarations that share the same scope - *scope related*. The next thing to check for in this stage of language analysis is *name conflicts*. When defining a data type or declaring an operation one must follow this rule:

**Rule 1.** *A name is the identifier assigned to a type definition or an operation declaration. It must be assigned uniquely within the same related component namespace.*

*Related components* are those of the same data type and operations. For example all the defined enumerated types are related and must be named differently. On the other hand a structure can be named the same as an operation. I'm aware that namespace rules can be considered as scoping rules, but nevertheless I've chosen to divide them into separate phases.

The constant expression syntactical construct consists of several expressions. Now we must specify the semantic rules for these expressions, so the first one:

**Rule 2.** *A constant expression must resolve to a primitive data type or to a previously defined constructed data type.*

Lets take a closer look at the constant expression syntactical grouping. The only places it can be found are the array declarator expression, attribute argument list and union switches and cases. Wire syntax supports arithmetic expressions, as well as relational expressions, logical expressions and bitwise expressions. These syntactical constructs are given a different name in the context of language semantics to provide more meaning. The semantics will refer to these constructs as *data type operations*. Every data type operation consists of *operators* and *operands*, and can only be applied

to certain data type. For example it makes no sense (semantically) to do a modulus operation on two strings, but on the other hand it does make sense to utilize the addition expression as a string concatenation operation. I've chosen not to allow to much freedom here and keep the semantics as simple as possible thus its rules easy to remember.

**Rule 3.** *Data type operation operands must be resolved to the same data type.*

This rule states that, for example, we can not add floating point numbers and integers. For integer data types all of the mentioned data type operations are defined as usual. Floating point types don't have the arithmetic modulus operation defined and bitwise operations. Of course it's clear that only the integer types can have defined bitwise operations. I've chosen to add string concatenation and furthermore to use the arithmetic addition expression to this purpose. Relational operations, at least equality and inequality, and logical operations are defined for every data type. The result of such an operation can be a logical *truth* or *false*. Wire users work with that specific abstraction and don't worry about the internal representation of this logical values.

Our abstract syntax tree is now fully checked for any semantical inconsistencies and given over to the next phase, code generation.

## 2.9. Wire code generation

The code generation subsystem receives a semantically valid abstract syntax tree. By traversing the tree it generates the corresponding code for the current node.

The generated code heavily relies on cross-platform Wire serialization/deserialization engine written in C, called *DeSer*. This engine provides a well define interface to handle serialization and deserialization of basic primitive data types: integers, floating point numbers and strings. It allows users to define the transfer syntax by specifying data alignment, byte and bit ordering, sizes, floating point representation, character encodings... This library relies on the *Bitstring* library for serialization and implements deserialization methods on its own. Lua extensions have also been implemented for interfacing with mentioned libraries from within Lua runtime environment.

The communication engine is not implemented, but basic use cases exists and the library interface is in the design phase. The decision lies on whether to implement this engine on top of *Berkley Socket* interface, provided on Linux systems, or to use an open-source, cross-platform and maintained solution. The main candidate is the *NSock*

project[2].

[2]Part of the Nmap project

# 3. Automatic recognition of network protocols

For starters let's further explain the chapter title. Classic approach to recognition of network protocols is to have a pattern database of known network protocols data units. This combined with a deterministic matching engine results in a system[1] for unambiguous recognition of network protocols.

This is where the *'automatic'* part comes in that completely differentiates pattern matching method from the one discussed in this thesis.

We're starting with the assumption that a network protocol assigned for recognition is yet unseen. Its specification is not publicly available. So our job is to somehow develop a system that could learn the structure of protocol data units and their exchange logic, ie. protocol operations.

One could say that, in a real world case scenario, this *'automatic'* method would follow the pattern matching method.

There are two scenarios for automatic protocol recognition:

**direct or active** The system is directly communicating with the target host. It uses some kind of request/response based algorithm to learn the protocol.

**proxy or passive** In this case the system is simply an observer that captures the communication of interest between two target hosts. Note that there is a lot more information exposed when using this method as the system obtains both the requests and responses.

There are analogies with the human communication. I call this the *Chinese* analogy. So for example the *direct* approach would consist of a target Chinese whose language I don't understand and me representing this learning system. I would *'talk'* to the Chinese hoping to get a response. Then by following certain heuristics I would

[1] Real world examples are Nmap network scanning engine and Amap application mapper

refine the way I talk, hopefully learning Chinese language. The *proxy* method would consist of me (learning system) listening to two Chinese talking to each other. Again heuristically learning their language.

From a practical point of view this two methods could further be divided into *on-line* and *off-line* methods, pertaining to the fact that the protocol data is collected real-time or captured and recorded for later processing, respectively.

The approach taken here for tackling the task of automatic protocol recognition is by using genetic algorithms. For that purpose a demonstrative implementation[2] has been developed using the *Evolutionary Computation Framework*.

## 3.1. Genetic algorithms overview

A *genetic algorithm* (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of *evolutionary algorithms* (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

**Algorithm 1** Genetic algorithm

$popsize \leftarrow DesiredPopulationSize$
$P \leftarrow \{\}$
**for** popsize times **do**
  $P \leftarrow P \cup \{NewRandomIndividual\}$
**end for**
$Best \leftarrow nil$
**repeat**
  $PNew \leftarrow \{\}$
  $Evaluate(P)$
  $PNew \leftarrow Select(P)$
  $Crossover(PNew)$
  $Mutate(Pnew)$
  $P \leftarrow PNew$
  $Best \leftarrow Best(P)$
**until** Termination criteria achieved
**return** Best

[2] A handy name of Babel Fish Project is given to this project

To use a genetic algorithm, you must represent a solution to your problem as a genome (or chromosome). The genetic algorithm then creates a population of solutions and applies genetic operators such as mutation and crossover to evolve the solutions in order to find the best one(s).

Instead of programming my own GA engine I decided on using - *Evolutionary Computation Framework*. ECF is a C++ framework intended for application of any type of evolutionary computation. It provides a handy evolutionary framework, including algorithms, genotypes and genetic operators. Also it has a solid XML based system for parameterization of your application. My sole occupation is to design a network protocol genotype and corresponding genetic operators (evaluation, crossover, mutation) and implement them in ECF.

## 3.2. Network protocol genotype

The method for automatic network protocol recognition developed in the scope of this thesis is the *proxy off-line* method, as described in the chapter 3. Further more, for practical reasons, I've limited the search space by reducing the problem to – *recognition of protocol data unit structuring*.

When developing a genotype first question you need to ask is: "How does an instance of a solution look like?". First of all protocol data must be captured and recorded. Once appropriately processed it's passed to the input of the learning system – *learning set*. Let's refer to this as *pdu instances*

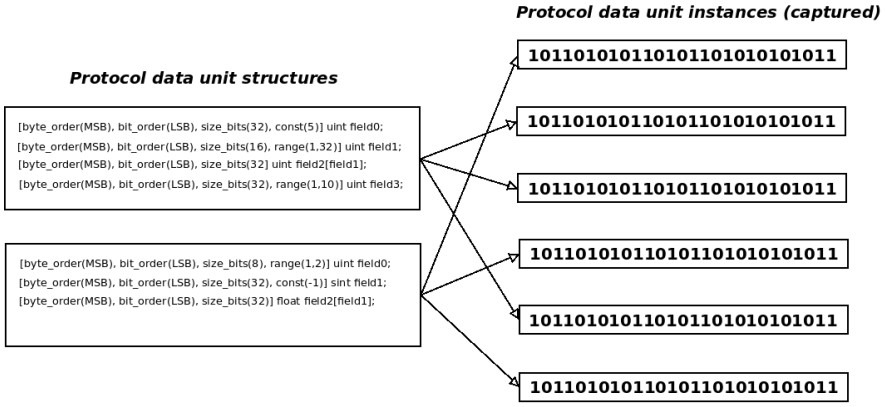


**Figure 3.1:** Network protocol genotype

Well we're trying to figure out the structuring of protocol data units passed to the input. Considering there can be more than one in a particular network protocol in-

stance, the solution consists of a number of *pdu structures*. Other than structuring, every pdu structure must have assigned an subset of input. This *pdu structure-pdu instances* relation tells us which part of input is *'described'* by which pdu structure.

The genotype consists (Figure 3.1) of pdu structures and every pdu structure is assigned a subset of pdu instances. A pdu structure is built of *fields*. A field can be of *uint, sint and float* types pertaining to unsigned and signed integers, and floating point numbers. Further more a field can be a sized array which size can be set constant or set by other fields in the same pdu structure.

Every field contains *attributes* which describes its transfer syntax and semantics. The defined attributes are:

**size_bits** Size of the element in bit measure.

**byte_order** Byte ordering for a byte sized field.

**bit_order** Bit ordering.

**range** Semantic attribute that defines a field value range.

**const** Semantic attribute that defines a field is constant in value.

Our goal is to learn the type of fields in a pdu structure and their attributes.

The network protocol genotype is implemented within ECF framework in *'NetProtoGen.cpp'* file. It has registered parameters for setting the maximal pdu structures number and of defining the number of captured data (pdu instances) to be considered a learning set:

**Listing 3.1:** Network protocol genotype parameters

```
1 <Genotype>
2         <NetProto>
3                 <Entry key="max_pdus">4</Entry>
4                 <Entry key="num_cap_pdus">1000</Entry>
5         </NetProto>
6 </Genotype>
```

## 3.3. Genetic operators

ECF provides all of the necessary core elements such as algorithm and fitness components. Our job is to develop three operators for our newly defined network protocol genotype. The first is the evaluation operator, then the crossover and mutation operators.

### 3.3.1. Evaluation

This was probably the most challenging part to implement. It consists of generating Lua code which performs the dissection of pdu instances according to a pdu structuring information held by the genotype. This code is invoked from the C++ environment and executed in Lua environment. Other than parsing it calculates the fitness for that genotype instance.

The fitness calculation is based on semantic validity of fields with assigned semantic attributes such as range, constant and array size semantic attributes. Note that the pdu structure size can, in general, be determined in runtime, so fitness calculation takes the size mismatch for every pair of pdu structure and pdu instance into consideration.

The constant and range attributes are checked for a field in a pdu structure and for every pdu instance assigned to this pdu structure. Any violation of these semantic restrictions is punished.

Two things can happen when parsing a field. First he field size can index data beyond a pdu instance size. Such occurrence must be punished by the evaluator. The second thing can indicate a valid data space inside the size boundaries of a pdu instance. This situation must be rewarded by the evaluator.

When the parsing of a single pdu instance is done according to structuring rules of a pdu structure, the size mismatch is calculated and punished by the evaluator.

The evaluation operator is implemented in *'NetProtoEvalOp.cpp'*. Its registered parameter sets the file name of the input file which contains the pdu instances.

### 3.3.2. Crossover

Crossover is a genetic operator used to combine genetic material of *parents* to produce a new individual, a *child*. A crossover operation represents a directed search component of genetic algorithms, oppose to a mutation operation which represents a random search component. By performing a crossover operation on two individuals we hope to explore the solution space near them, hopefully finding a better solution.

There are two step in recombining network protocol genotypes. The first takes two random pdu structures from both parents, and performs a sort of one cross-point crossover with points represented as fields. The second operation deals with pdu instance assignments. If a genotype has the same number of pdu structures then the pdu instance assignments are copied to a child from randomly chosen parent.

Crossover is implemented in *'NetProtoCrxOp.cpp'* and it has no registered parameters.

### 3.3.3. Mutation

This is the simplest operation, and you can get pretty creative when designing a mutation operation. A network protocol genotype mutation operator also operates in two phases. The first randomly chooses a pdu structure and rebuilds it from scratch and the second resets the pdu instance assignments.

Mutation is implemented in *'NetProtoMutOp.cpp'* and it has no registered parameters.

# 4. Conclusion

Time invested in designing and developing the *Wire* language for network protocol specification resulted in clear definitions of language purpose, lexical conventions, syntax and semantics. A cross-platform serialization engine has been developed (and ported to Lua) in C on which the generated code is heavily dependent. A priority is the implementation of a Wire communication engine, or integration with existing open-source solutions. Future works consists of making a Python based code generation plug-in architecture, for more practical and easier code generation. Furthermore the GNU M4 macro language is to be utilized for implementation of Wire code inclusions.

The proposition that the task of automatic network protocol data units recognition can be solved using the genetic algorithms has proven faulty. Even in theoretical considerations the choice of using meta-heuristic search methods is wrong. The reason being that protocol data unit fields are, most commonly, mutually independent and lack the semantic relationships. Therefore the genetic evaluator has no way, or little way, of determining a fitness of an individual. Nevertheless a network protocol genotype has been developed using the C++ ECF framework, including the genetic operators (evaluator, crossover, mutation). The test example included 'recognition' of *Internet Protocol* (IP) data units.

**Abstract**

Thesis gives an overview on network protocol theory including protocol design, specification and implementation. A network protocol specification language called Wire has been developed in the scope of this thesis. Detailed descriptions on the analysis of the Wire language are given, as well as on code generation. An overview of Wire language is provided using an example network protocol.

The problem of automatic network protocol recognition has been addressed in the scope of this thesis. Genetic algorithms have been utilized for solving this problem, therefore a network protocol genotype and corresponding genetic operators have been developed and implemented using the C++ Evolutionary Computation Framework.

**Keywords:** network protocol theory, abstract syntax notation, wire definition language, automatic recognition, genetic algorithms, evolutionary computation framework

**Metode predstavljanja i automatskog prepoznavanja mrežnih protokola**

**Sažetak**

Rad daje pregled na teorijom mrežnih protokola, uključujući dizajn, specifikaciju i implementaciju mrežnih protokola. U sklopu rada ostvaren je jezik za specifikaciju mrežnih protokola nazvan Wire. Analiza jezika Wire i stvaranje koda su detaljno objašnjeni. Napravljen je pregled nad jezikom Wire koristeći primjerni mrežni protokol.

Problem automatskog prepoznavanja mrežnih protokola također se obrađuje u sklopu ovog rada. Za rješavanje tog problema korišteni su genetski algoritmi, stoga su razvijeni genotip za predstavljanje mrežnog protokola i odgovarajući genetski operatori koristeći C++ okruženje Evolutionary Computation Framework.

**Ključne riječi:** teorija mrežnih protokola, notacija za apstraktnu sintaksu, wire jezik, automatsko prepoznavanje, genetski algoritmi, evolutionary computation framework

# BIBLIOGRAPHY


[1] Andrew S. Tanenbaum, *Computer Networks*. Prentice Hall, 4nd Edition, 2003.

[2] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison Wesley, 2003.

[3] ITU-T, *Open System Interconnection - Basic Reference Model*. ITU, 1994.

[4] John Larmouth, *ASN.1 Complete*. Open Systems Solutions, 1999.

[5] Olivier Dubuois, *ASN.1 - Communication between Heterogeneous Systems*. OSS Nokalva, 2000.

[6] Charles Donnelly and Richard Stallman, *Bison - The Yacc-compatible Parser Generator*. Free Software Foundation 51 Franklin Street, Fifth Floor Boston, MA 02110-1301 USA, 2009.

[7] John Levine, *Flex And Bison*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. 2009.

[8] Kurt Jung and Aaron Brown, *Beginning Lua Programming*. Wiley Publishing, Inc., Indianapolis, Indiana 2007.

[9] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *Lua 5.1 Reference Manual*, Lua.org, August 2006.

[10] Marin Golub, *Genetski algoritam, Prvi dio*, FER, 2010.

# Appendix A
# Wire lexical definitions

**Listing A.1:** wire.l

```
%option nodefault noyywrap yylineno

%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "wire_utils.h" //debug, error
#include "wire_lex.h" //lex utils
#include "wire_ast.h" //so wire.tab.h has a definition of pnode_t
#include "wire.tab.h" //tokens

int yyparse();

#define YYDEBUG 1
%}

reCOMMENT          ("/*"([^"*"]|[\r\n]|("*"+([^"*/"]|[\r\n])))*\*+\/)|("//".*)|

reNEWLINE ([\r?\n])

reWHITESPACE ([\t\f ]+|{reNEWLINE})

reIDENTIFIER ([A-Za-z_][A-Za-z0-9_]*)

reINTCONST ({reHEXCONST}|{reBINCONST}|{reOCTCONST}|{reDECCONST})

```

```
27 reHEXCONST (0(x|X)[0-9a-fA-F]+)
28
29 reBINCONST (0(b|B)[01]+)
30
31 reOCTCONST (0[0-7]+)
32
33 reDECCONST ([0-9][0-9]*)
34
35 reFLOATCONST ([0-9]*\.[0-9]+([eE][-+]?[0-9]+)?)
36
37 reSTRINGCONST (\"([\40-\41\43-\176])*\")
38
39 %%
40 {reINTCONST} {
41 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣yylval.text = strdup(yytext);
42 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣print_debug("INT_CONST: %s\n",yylval.text);
43 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return tINTCONST;
44 ␣␣␣␣␣␣␣␣}
45
46 {reFLOATCONST} {
47 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣yylval.text = strdup(yytext);
48 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣print_debug("REAL_CONST: %s\n",yylval.text);
49 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return tFLOATCONST;
50 ␣␣␣␣␣␣␣␣}
51
52 {reSTRINGCONST} {
53 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣yylval.text = strdup(yytext);
54 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣print_debug("STRING_CONST: %s\n",yylval.text);
55 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return tSTRINGCONST;
56 ␣␣␣␣␣␣␣␣}
57
58 {reIDENTIFIER} {
59 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣yylval.text = strdup(yytext);
60 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣print_debug("IDENTIFIER: %s\n",yylval.text);
61 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return get_token_by_identifier(yytext);
62 ␣␣␣␣␣␣␣␣}
63
64 "==" {
```

```
65 ␣␣␣␣␣␣␣␣print_debug("RELATIONAL OP: %s\n",yylval.text);
66 ␣␣␣␣␣␣␣␣return␣tRELEQU;
67 ␣␣␣␣␣␣␣␣}
68
69 "!="␣{
70 ␣␣␣␣␣␣␣␣print_debug("RELATIONAL OP: %s\n",yylval.text);
71 ␣␣␣␣␣␣␣␣return␣tRELNEQU;
72 ␣␣␣␣␣␣␣␣}
73
74 ">="␣{
75 ␣␣␣␣␣␣␣␣print_debug("RELATIONAL OP: %s\n",yylval.text);
76 ␣␣␣␣␣␣␣␣return␣tRELGE;
77 ␣␣␣␣␣␣␣␣}
78
79 "<="␣{
80 ␣␣␣␣␣␣␣␣print_debug("RELATIONAL OP: %s\n",yylval.text);
81 ␣␣␣␣␣␣␣␣return␣tRELLE;
82 ␣␣␣␣␣␣␣␣}
83
84 "&&"␣{
85 ␣␣␣␣␣␣␣␣print_debug("LOGICAL OP: %s\n",yylval.text);
86 ␣␣␣␣␣␣␣␣return␣tLOGAND;
87 ␣␣␣␣␣␣␣␣}
88
89 "||"␣{
90 ␣␣␣␣␣␣␣␣print_debug("LOGICAL OP: %s\n",yylval.text);
91 ␣␣␣␣␣␣␣␣return␣tLOGOR;
92 ␣␣␣␣␣␣␣␣}
93
94 "<<"␣{
95 ␣␣␣␣␣␣␣␣print_debug("BITWISE OP: %s\n",yylval.text);
96 ␣␣␣␣␣␣␣␣return␣tBITSL;
97 ␣␣␣␣␣␣␣␣}
98
99 ">>"␣{
100 ␣␣␣␣␣␣␣␣print_debug("BITWISE OP: %s\n",yylval.text);
101 ␣␣␣␣␣␣␣␣return␣tBITSR;
102 ␣␣␣␣␣␣␣␣}
```

```
103
104 ["\[\](){}%/*+\-;,=&|\^<>:!."] {
105                     print_debug("OP: %c\n", *yytext);
106                     return *yytext;
107             }
108
109 {reWHITESPACE}    ;
110
111 .           {
112                     print_error("%s <%s> -- line %d","invalid character", yytext
113                     exit(1);
114             }
115
116
117 %%
118 int main(int argc, char* argv[])
119 {
120             yyin = fopen(argv[argc-1], "r" );
121       if(yyin == NULL)
122       {
123                     perror("fopen");
124                     return 2;
125       }
126
127             yyparse();
128             return 0;
129 }
```

# Appendix B
# Wire syntax definitions

**Listing B.1:** Wire BNF grammar listing produced by GNU Bison report mechanism

```
Grammar

    0 $accept: wire $end

    1 wire: protocol
    2     | /* empty */

    3 protocol: attribute_list_opt tPROTOCOL tIDENTIFIER '{' protocol_body_opt '}'

    4 protocol_body_opt: protocol_body
    5                  | /* empty */

    6 protocol_body: protocol_body_component ';'
    7              | protocol_body protocol_body_component

    8 protocol_body_component: type_definition
    9                        | operation_declarator

   10 attribute_list_opt: '[' attribute_list ']'
   11                   | /* empty */

   12 attribute_list: attribute
   13               | attribute_list ',' attribute

   14 attribute: tIDENTIFIER
   15          | tIDENTIFIER '(' attribute_argument_list ')'

   16 attribute_argument_list: attribute_argument
   17                        | attribute_argument_list ',' attribute_argument

   18 attribute_argument: const_exp

   19 type_definition: enum_definition
   20                | union_definition
   21                | struct_definition
   22                | pdu_definition

   23 enum_definition: attribute_list_opt tENUM tIDENTIFIER '{' enum_body '}'

   24 enum_body: enum_body_component
   25          | enum_body ',' enum_body_component

   26 enum_body_component: tIDENTIFIER '=' const_exp
   27                    | tIDENTIFIER

   28 union_definition: attribute_list_opt tUNION tIDENTIFIER '{' union_body '}'

   29 union_body: union_body_component
   30           | union_body union_body_component
```

```
31 union_body_component: const_exp ':' local_declarator_list
32                     | tDEFAULT ':' local_declarator_list
33
34 struct_definition: attribute_list_opt tSTRUCT tIDENTIFIER '{' struct_body '}'
35
36 struct_body: struct_body_component
37           | struct_body struct_body_component
38
39 struct_body_component: local_declarator ';'
40
41 pdu_definition: attribute_list_opt tPDU tIDENTIFIER '{' pdu_body '}'
42
43 pdu_body: pdu_body_component
44         | pdu_body pdu_body_component
45
46 pdu_body_component: local_declarator ';'
47
48 local_declarator: primitive_local_declarator
49                 | constructed_local_declarator
50                 | anon_local_declarator
51
52 local_declarator_list: local_declarator ';'
53                      | local_declarator_list local_declarator ';'
54
55 primitive_local_declarator: attribute_list_opt tBYTE tIDENTIFIER array_declarator_opt
56                           | attribute_list_opt tFLOAT tIDENTIFIER array_declarator_opt
57                           | attribute_list_opt tSTRING tIDENTIFIER array_declarator_opt
58                           | attribute_list_opt tUINT tIDENTIFIER array_declarator_opt
59                           | attribute_list_opt tSINT tIDENTIFIER array_declarator_opt
60
61 constructed_local_declarator: attribute_list_opt tENUM tIDENTIFIER tIDENTIFIER array_declarator_opt
62                             | attribute_list_opt tSTRUCT tIDENTIFIER tIDENTIFIER array_declarator_opt
63                             | attribute_list_opt tUNION tIDENTIFIER tIDENTIFIER array_declarator_opt
64                             | attribute_list_opt tPDU tIDENTIFIER tIDENTIFIER array_declarator_opt
65
66 anon_local_declarator: attribute_list_opt tUNION '<' const_exp '>' '{' union_body '}'
67
68 array_declarator_opt: '[' const_exp ']'
69                     | /* empty */
70
71 operation_declarator: attribute_list_opt tOPERATION tIDENTIFIER '(' operation_arg_list ')'
72
73 operation_arg_list: operation_arg ','
74                   | operation_arg_list operation_arg
75
76 operation_arg: attribute_list tPDU tIDENTIFIER tIDENTIFIER
77
78 const_exp: integer_const_exp
79          | float_const_exp
80          | string_const_exp
81          | identifier
82          | arithmetic_exp
83          | relational_exp
84          | logical_exp
85          | bitwise_exp
86
87 float_const_exp: tFLOATCONST
88
89 string_const_exp: tSTRINGCONST
90
91 integer_const_exp: tINTCONST
92
93 arithmetic_exp: const_exp '+' const_exp
94               | const_exp '-' const_exp
95               | const_exp '*' const_exp
96               | const_exp '/' const_exp
97               | const_exp '%' const_exp
98
99 relational_exp: const_exp '>' const_exp
100              | const_exp '<' const_exp
101              | const_exp tRELEQU const_exp
102              | const_exp tRELNEQU const_exp
103              | const_exp tRELGE const_exp
```

```
124     83                | const_exp tRELLE const_exp
125
126     84 logical_exp: '!' const_exp
127     85              | const_exp tLOGAND const_exp
128     86              | const_exp tLOGOR const_exp
129
130     87 bitwise_exp: '~' const_exp
131     88              | const_exp '&' const_exp
132     89              | const_exp '|' const_exp
133     90              | const_exp '^' const_exp
134     91              | const_exp tBITSR const_exp
135     92              | const_exp tBITSL const_exp
136
137     93 identifier: tIDENTIFIER
138     94           | identifier '.' tIDENTIFIER
```