COMPRION

Diploma Thesis

Specification Language for Binary Parser Construction in the Context of Smart Card Protocol Monitoring

Jürgen Wall

presented to

Prof. Dr. Engels
Department for Database- and Informationsystems

and

Prof. Dr. Kastens
Programming Languages and Compilers

Faculty of
Electrical Engineering, Computer Sciences and Mathematics

University of Paderborn
Warburger Straße 100
D-33095 Paderborn

31.03.2010

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe, sowie ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Paderborn, den 31.03.2010

_____

Jürgen Wall

Contents

1 Introduction

The evolution of protocols in communication technologies is a rapid process today. Driven by standardization efforts, manufacturers of communication devices heavily rely on computer-aided validation of standard compliance of their products. This applies in particular to protocol implementations, whose correctness is essential for interoperability. However, due to the volume and complexity of normative documents to cover the development and maintenance of test software for automated communication validation is a comprehensive task.

The declarative and often informal specifications of protocol communication packets and their low-level data structures represent a noticeable hurdle for a manual transcription into an operational protocol analysis implementation. The task requires not only understanding of the specification document and the technological background, but also needs reliable low-level programming skills, in particular for the dissection of structured binary data. This is known to be a time-consuming process and a source of frequently recurring programming errors.
Hence, in this work we discuss a systematic approach to leverage the construction of binary data parsers. We present the Paderborn Binary Parser Generator (PBPG), a formal specification language and its compiler for automatic derivation of operational parsing code from formalized data structure specifications in PBPG syntax.

Founding on composition of C-like structs, unions and arrays from elementary data types in combination with semantic constraints PBPG bridges the semantic gap between the declarative form of data structure specifications and the corresponding operational parser code.

The generated binary parsers are well suited for the rapid development of protocol monitoring and analysis applications by avoiding errors of manual implementation and reducing the amount of development time.

1.1 Motivation

To ensure a high level of interoperability among the multitude of devices, the telecom industry heavily relies on standards. Being developed and maintained by standardization organizations such as ISO, IEC, ETSI and 3GPP, those standards establish a basis for the development of reliable and interoperable technology.

COMPRION as a test equipment manufacturer provides the hardware and software for the standard compliance verification of mobile devices. Unlike handset and card manufacturers, which typically concentrate on coverage of standards relevant to a certain product, the test equipment manufacturer must gear up for various configurations of interacting devices, potentially requiring a comprehensive batch of test scenario implementations. In order to keep the test equipment conforming to the latest standard, frequent updates of legacy system parts must be performed. This applies in particular to communication monitoring tools.

The purpose of a monitoring application by COMPRION is to display the smart card-terminal interaction in a human readable way, for the inspection and debugging of either of the device implementations. To accomplish this, the application must dissect the binary data stream and extract the embedded information elements. Thereafter the gained information is presented with textual annotations.

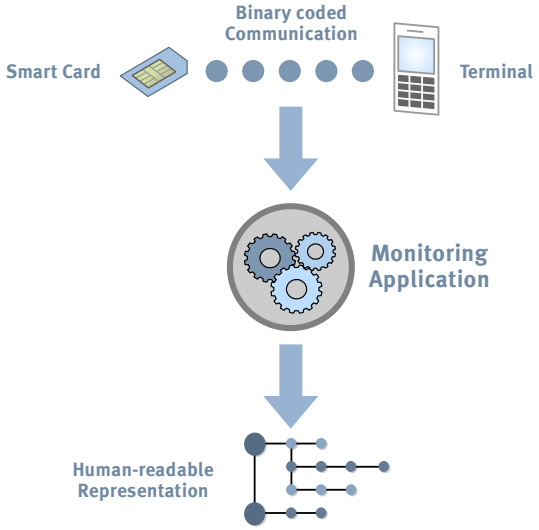


Figure 1.1: *The monitoring application dissects the binary coded data stream into its constituting elements and generates a readable representation of the data.*

For the dissection of raw communication data, the application must implement parts of the employed communication protocol and the transferred data objects. As long as the test system and the mobile devices follow the same specification, correct dissection can be expected. However, due to the evolution in communication standards and the communicating devices, the test system may become inconsistent with its tes-

tees. This puts particular importance on the actuality of the dissection procedures of the monitoring application.

Currently, modification and extension of the dissection procedures is done by manually transcribing structure specifications from standard documents to operational program logic. However, due to the declarative nature of standard documents, mapping of the semantics to an imperative programming language is a tedious and error-prone process. Moreover, handling low-level protocol data by means of a high level programming language is sometimes inconvenient, which complicates and retards implementation and maintenance.

But, considering input and output of a monitoring application as depicted in figure 1.1 as sentences of two distinct languages, the process of monitoring can be interpreted as a translation task. With the machine-oriented language as source and human readable representation as target language, this problem is fairly well suited to translation techniques of compiler theory.
Hence, the goals of this thesis were to work out the differences between parsing textual and binary inputs and to analyze the applicability of the traditional techniques. As a practical result, the design and development of a declarative specification language and its compiler for the generation of binary parsers were aimed. Striving for increased productivity and reliability, not only the quality of binary parsers, but also the quality of the development process itself were in the focus of this thesis.

1.2 Document Structure

This document is organized as follows. Starting with foundations in clause 2 we introduce basic aspects of smart cards, their internal organization and a typical monitoring system set-up. After discussing basic notions and techniques of compiler theory, we focus on processing of binary input and a modification to the analysis model. We conclude the foundations clause with a theoretical approach to parsing binary data.

In clause 3 we expose related projects in the field of protocol monitoring and analysis, which have slightly different goals but rely on linguistic approaches. In the subsequent clause 4.1 we work out quality requirements for a reasonable solution in the context of this work.

Thereafter we present in clause 4 the design of our solution (PBPG), which includes the language, its compiler and the model of a binary parser. There we focus on the architecture, concepts and the choices made with regard to the tools employed.

Clause 5 gives then a closer view on some of the interesting aspects and computational difficulties encountered during the development of PBPG.

Finally, in clause 6 the developed product is evaluated against the stated quality requirements from clause 4.1 and a conclusion along with an outlook is given in clause 7.

2 Foundations

In this section we are going to introduce the reader to the field of smart cards and the binary communication on the interface between a terminal and a smart card which is the pivotal subject of our focus. Our aim to utilize compiler construction techniques for the generation of parsers for the analysis of binary coded communication streams leads us across a couple of topics and subjects which can fill books on their own. With a rather selective point of view, we are going to inspect the following topics in the next clauses.

Starting in clause 2.1 we introduce the smart card with today's standardized communication protocols and the concept of layered communication. We outline the logical organization of files on the card and the set of commands used to control the card connected to a terminal.

Moving on to clause 2.2 we describe the process of monitoring the multi-layer communication for the purpose of standard compliance verification as performed by COMPRION. We analyze the manually implemented protocol packet parsing and identify drawbacks associated with this approach.

In clause 2.3 we review aspects of the compilation process with an emphasis on analysis. This encloses the individual processing stages and the involved computation models in association with properties of grammars and languages.

In clause 2.4 we argue about modifications of the parsing process, which become necessary when applied to binary encoded languages. We try to outline the associated consequences for the complexity of the analysis and the form of binary structure specifications. We conclude the section with a short summary.

In each clause we cite the literature used for the very topic instantly at the position required. The literature used for the subject area of smart cards consists of the normative documents and especially the *"Smart Card Handbook"* by Rankl and Effing. Major contribution to the topic of compiler theory has been made by *"Compilers: Principles, Techniques, and Tools"* alias *"The Dragonbook"* by Aho, Lam, Sethi and Ullman [ALSU06], *"Übersetzerbau"* by Kastens [Kas90] and *"Engineering a Compiler"* by Cooper and Torczon.

2.1 Smart Cards

Today a large number of different smart cards, developed for distinct purposes exists. Initially introduced for electronic payment, smart cards gained increasing importance in mobile communication technology, health care information systems and authentication. Its success is originated in the presence of a microprocessor[1] and the high memory capacities compared to other chip cards [RE03]. The ability to securely protect data from unauthorized access and to execute cryptographic algorithms, makes smart cards especially applicable in security related areas.

Depending on the standard, smart cards of mobile technology are denoted as *Subscriber Identity Module* (SIM) or as *Universal Integrated Circuit Card* (UICC). The former term originates from the GSM standard [ETS95], which is the most widespread technology today. The second term is introduced by the ETSI TS 102.221 [Eur09] standard, that supersedes GSM, and defines GSM-based operation as an integrated application besides others. We appoint to keep using the general term *smart card*, since the ongoing considerations can easily be applied to multiple concrete systems.

Based on these standards the communication protocols of the smart card interface are shown in the next clause. Thereafter the organization of a smart card and the defined operations are introduced, providing the foundations for ongoing discussion.

2.1.1 Protocols

The ISO Open System Interconnection (OSI) Basic Reference Model [ISO94] is the foundation for many communication protocols. It defines communication between two or more systems by means of a stack of abstraction layers. This modular concept assigns every layer a specific function, that can be implemented by one or more concrete protocols. Each layer employs custom units of interaction, with granularities proportional to the layer number. Units of lower layers encapsulate unit-segments of adjacent layers above (*protocol encapsulation*). This makes protocols exchangeable, without affecting the overall functionality. Therefore, the higher layer protocols can rely on the protocol stack below without caring about lower level details of connection.
Seven layers in total are defined, though not all of them are applicable to concrete communication systems. This also applies to communication over the smart card interface, which is presented here. For the complete description of the Basic Reference Model please refer to its specification document [ISO94].

Compared to the reference model the organization of the smart card communication model [Eur09] is less complex. The layer stack comprises four layers only[2], omitting the routing facility located at L3, not relevant to this technology. Furthermore, the layers five to seven are merged to a single *application layer*.

[1] The term *smart* reflects the card's ability to calculate.

[2] In literature the *transport layer* is sometimes merged with *data link layer*, still being called *layer 2*, which sums up to three layers in total.

Layer 1 and 2: Let us consider the *physical layer* first. In most smart cards used today, according to the contact assignments of ISO 7816-2 [ISOa], only one of eight available electrical contacts is used for data transmission, whereas other contacts are reserved for power supply, system reset and other operations. That means, that only one peer can send data at a time, while the other waits for receipt. From this *synchronous* transmission the signal-encoded bits are assembled to *frames* at the *data link layer*. Frames hold a single byte of data with additional control bits.

Layer 3: After assembly those *frames* are passed to protocols of the *transport layer* L4. The most prevalent protocols here are the T=0 and T=1 transmission protocols[3], whereas other alternatives [ISOa, ISOb, RE02, RE03] exist. Both of them are capable of serving the same ISO *transport layer* [Eur09], though differing in the exact process. Out of *frames*, both protocols construct Transfer Protocol Data Units (TPDUs), which represent *segments* in terms of the OSI model. To preserve communication quality for the *application layer*, retransmission of corrupt and missing TPDUs is managed in this layer.
Two variants of TPDUs are specified - Command TPDU (C-TPDU) and corresponding Response TPDU (R-TPDU). The first is emitted by the terminal as a command, while the latter is emitted by the card in response. Lasting for the entire communication session, this command-response interplay realizes the *master-slave* principle, with the terminal as master.

[3]The names might seem unusual. They are spoken out "t equals zero" and "t equals one".

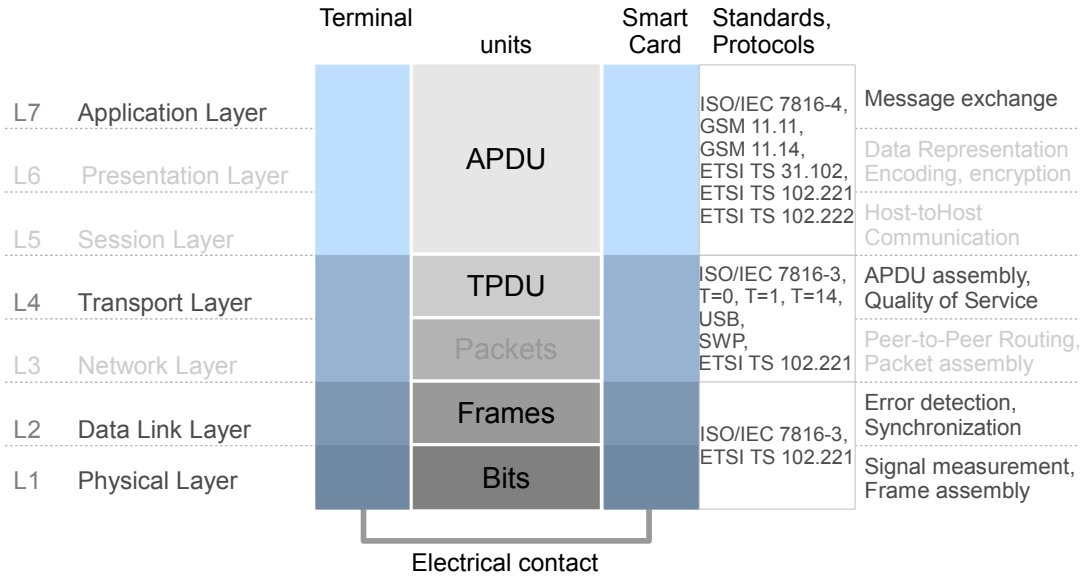


Figure 2.1: *The smart card communication requires less layers than the OSI reference model. The omitted layers are grayed out. In the Standards-column specification documents are referenced, which define protocols for the specific layer.*

Layer 4: Finally, the TPDUs are combined to larger Application Protocol Data Units (APDUs) of the *application layer*. C-TPDUs contribute segments of a C-APDU and R-TPDUs the segments of a R-APDU. At this level, the communication is guided by the order of *messages* and the encoding of their payload. This layer serves the application logic by transporting application specific data, which encloses control data of superior protocols and stored file contents. A vast number of specialized data structures is defined here, which produces the majority of implementation effort for a protocol monitoring application. Therefore, the application layer is most important for this thesis.

Summarizing this clause, the general information to keep in mind is, that communication data is typically transferred in hierarchically constructed units and that for each layer one or more protocols are defined to accomplish a layer-specific task. Specifications describe the concrete *encoding*, the order and meaning of the protocol data units, i.e. the *syntax* and *semantics* of the machine-machine interaction. Since the application layer exhibits a rich variety of data structures, it offers the highest application potential for binary parsers.

2.1.2 Logical Organization

Smart cards exhibit a simple file system, resembling those of common operating systems. The root denoted as Master File (MF), encloses the entire storage memory of the card, including all other files and directories. Below the MF three different kinds of files can be stored, Elementary Files (EFs), Dedicated Files (DFs) and Application Directory Files (ADFs) as shown in figure 2.2. They are referenced through a two-byte logical File Identifier (FID). The MF is identified by the unique FID '3F00'. Except 'FFFF' - which is reserved for future use - and the MF FID, all 16-Bit values are available for application use [ISOc], i.e. are standardized accordingly.

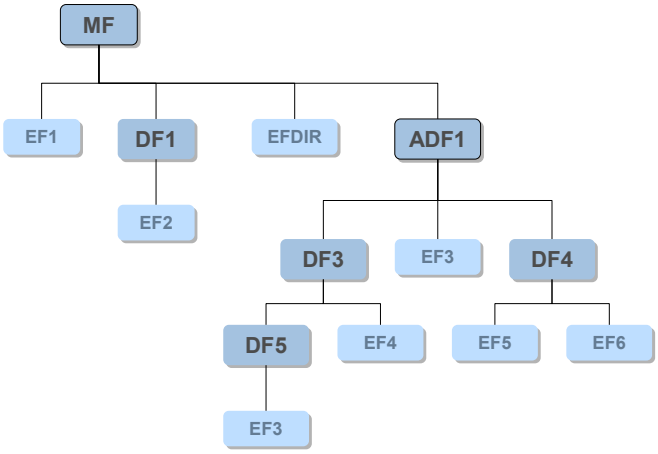


Figure 2.2: *Example of a logical structure, taken from [Eur09].*

Elementary File (EF) is a file, which can contain data, but no other files. In its simplest form its structure is not restricted by the file system and the EF contents are totally organized by the owning application. This type of EF is denoted as "Transparent". Other available types are "Linear fixed" and "Linear variable", which are file system handled record lists, with either a *fixed* or a *variable* length for each record. The records of a Linear EF, can be addressed through an absolute record number, or relatively, by referencing adjacent records of a recently addressed record. Finally, such lists can be circularly linked, in order to allow looping through the records of a file. This results in a "Cyclic" file type which allows for convenient handling of large datasets as used for phone books and schedule lists for example. The structure and contents of a record are of course not restricted by the file system.

Dedicated File (DF) represents a folder, possibly containing further EFs or DFs. The DF nesting depth is unrestricted in general, however typical applications do not exceed a depth of two. To prevent name collisions, DFs can be accessed alternatively by their "DF name", which is a 1 to 16 bytes long identifier. The "DF name" in combination with an Application Identifier (AID) of a smart card application, is a globally unique denotation of directories.

Application Directory File (ADF) is a special form of DF, representing a smart card application. Applications were introduced by ETSI as a generalizing extension of the GSM technology. ADFs are supposed to enclose all files required for their execution, thus allowing to place multiple applications below the MF side by side. A single Universal Integrated Circuit Card (UICC) may thus integrate applications to access different communication networks or even financial functionality. ADFs are identified by their AID, which has for registered applications a globally unique value. The first access of an ADF is done via the AID, and is typically associated with a user PIN request. All ADFs on the smart card are listed in a DF called "DF_{DIR}" under the MF.

2.1.3 Smart Card Commands

In clause 2.1.1 we introduced the C-APDU as a command, a terminal can issue to the card. The specific commands, accepted by the cards can vary, depending on the implemented standard. Smart cards complying to ISO/IEC 7816-4 [ISOc] provide a small API for file management, general operation and authentication. Not all cards have to support all of the commands, or all possible parameter combinations regarding the ISO/IEC 7816-4.

In the following tables the general formats of the C-APDU and the R-APDU are shown. Both form containers for the transferred application data enclosed by control data. The values and parameters of the concrete APDUs are instances of the general format. Some of the smart card commands will be shown in detail on the next pages.

A C-APDU can have multiple forms differing in the encoding of the body. The C-APDU header is mandatory and therefore always present. It starts with a command

Field	Length	Description	Grouping
CLA	1	Class of instruction	Header
INS	1	Instruction code	
P1	1	Instruction parameter 1	
P2	1	Instruction parameter 2	
Lc	0 or 1	Number of bytes in the command data field Body	Body
Data	Lc	Command data string	
Le	0 or 1	Max. number of data bytes expected in response of the command	

Table 2.1: *Contents of a C-APDU specified in ETSI TS 102221 [Eur09].*

Case	Structure
1	CLA INS P1 P2
2	CLA INS P1 P2 Le
3	CLA INS P1 P2 Lc Data
4	CLA INS P1 P2 Lc Data Le

Table 2.2: *Cases of C-APDUs.*

class indication and is used for distinguishing the underlying technology[4]. The second field is the instruction code, which indicates what operation the smart card has to perform. A trailing pair of instruction parameters encodes additional instruction-dependent information.

In contrast to the command header, the structure of the body is not static and depends on the values encoded in the header. In its simplest form, the body can be empty. In such a case, the size of the APDU depends on the header size solely. If and only if the body consists of a single byte, it is to be interpreted as the length of the Response APDU (R-APDU) the terminal expects from the card. Otherwise, the first byte of body represents the length of the consecutive data field. If there is one more byte to come after the data field, it is again to be interpreted as expected response length, though it might be absent as well.

Code	Length	Description
Data	Lr	Response data string
SW1	1	Status byte 1
SW2	1	Status byte 2

Table 2.3: *Contents of R-APDU*

The format of a R-APDU is somewhat simpler. It contains an optional *Data* field, whose length depends on the concrete command and the requested information to be returned. The succeeding *status* field is mandatory. It consists of two bytes SW1 and

[4]GSM commands have different classes than the UICC or CDMA commands

SW2, encoding an execution success indicator. Possible values and their interpretation depend on the commands, but the value indicating successful execution is always defined as SW1 = 0x90, SW2 = 0x00.

Based on this knowledge we now explain briefly some of the frequently used commands. In the example we use the GSM technology class byte (0xA0).

SELECT The SELECT command allows to mark a file as the *selected* file of the file system. Succeeding READ and WRITE commands are related to this very file. Mostly the FID of the file to select is stated in the data field of the command body, however, also absolute and relative *paths* or DFs names are accepted as arguments. The card's response to the SELECT command may contain the file size, access conditions and proprietary information.

CLA	INS	P1	P2	Lc	Data	Le
0xA0	0xA4	0x00	0x00	0x02	File ID	0x00

Table 2.4: *Coding of SELECT Command*

VERIFY PIN This command compares the PIN entered by the user with the value stored on the smart card. Such verification is required to restrict access to secured files and applications. Since the security mechanism allows to maintain multiple PINs, the parameter P2 accepts a reference to the PIN to compare the user value with. The data field holds the user entered PIN. In case of successful comparison, the card responds with a SW1 = 0x90, SW2 = 0x00 and the access to the restricted contents is granted to the user. In case of failed comparison, the card returns a status of SW1 = 0x63 and SW2 = 0xCX, where X is the number of remaining comparison trials. After three failed trials the PIN is blocked.

CLA	INS	P1	P2	Lc	Data
0xA0	0x20	0x00	b5..b1: PIN Nummer	0x08	PIN (8 Byte)

Table 2.5: *Coding of VERIFY PIN Command*

READ BINARY After selecting a "Transparent" file using SELECT, the READ BINARY command allows to read its contents. The card can only perform this operation, if the PIN associated with the file was verified successfully. If the execution succeeds, the R-APDU contains file data starting at the offset described in P1 and P2, with the length of Le bytes.

CLA	INS	P1	P2	Le
0xA0	0xB0	b7..b1: Offset High	Offset Low	Length of Response

Table 2.6: *Coding of READ BINARY command*

UPDATE BINARY After selecting a "Transparent" file using SELECT, the UPDATE BINARY command allows to (over)write its contents. Same as for reading, writing of a file can only be performed after successful authentication. P1 and P2 form the offset in the file to place the data field value at.

CLA	INS	P1	P2	Lc	Data
0xA0	0xD6	b7..b1: Offset High	Offset Low	Length of Data	Value

Table 2.7: *Coding of UPDATE BINARY Command*

The complete list of operations standardized by the ISO/IEC 7816-4 consists of 18 commands and can be found in the annex on page 85. For technical details and further information please refer to the normative document [ISOc] or to the "Smart Card Handbook" [RE03].

2.2 Monitoring

Since the layered smart card communication is inaccessible to the human in a direct way, a monitoring application is required to inform the user about the data flow and the device states. It helps in identifying malfunction of either of two systems and aids the manufacturers in the development of new products. On the next pages we are going to present the current approach of monitoring binary communication at COMPRION. Thereafter we will depict the drawbacks of the current implementation, preparing arguments for the conception of a new approach.

2.2.1 Technical Setup

The test system consists of a tracing hardware device (*tracer*), a computer system with Microsoft Windows and the monitoring software installed. The tracer is connected to the computer via a USB wire, offering interfaces for the connection of a terminal and a smart card as shown in figure 2.3.

Figure 2.3: *Monitoring of smart card communication using a COMPRION monitoring hardware. The traced data stream is passed to the monitoring application residing on a computer, where the parsing, monitoring and further analysis are performed.*

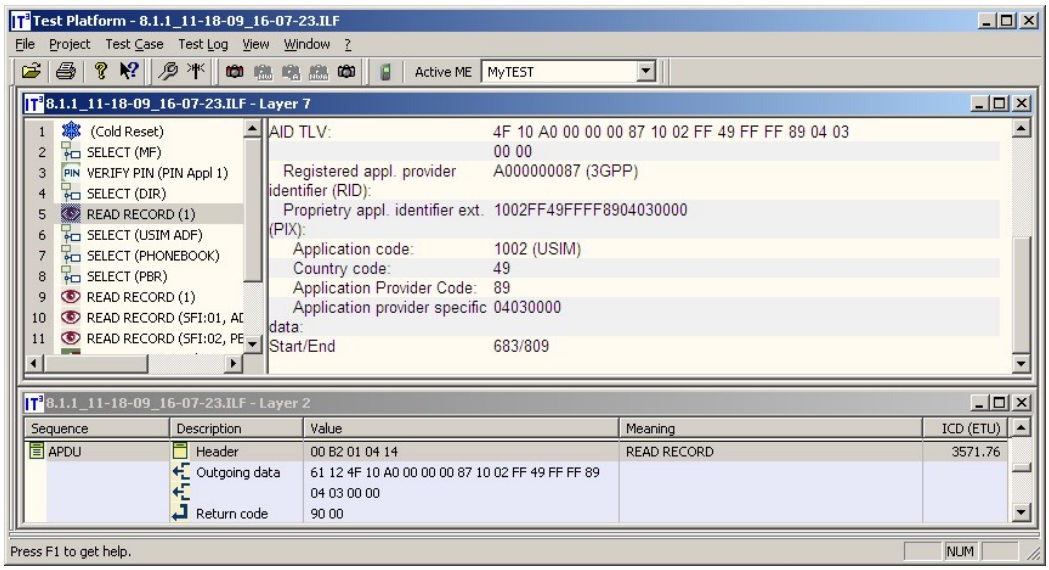


Figure 2.4: *The monitoring application displays parsed contents of the transport layer and the application layer.*

In the tracer hardware processing of the physical layer and the data link layer of the protocol stack is implemented. The monitoring application on the other hand implements the transport layer and the application layer, and is therefore responsible for the assembly of APDUs out of the frame-stream from the hardware. The user is offered a textual representation of the three digital layers annotated with descriptive terms, designating the values and data structures. In case of erroneous or unknown data, the parsing mechanism falls back to a raw-output state and prints the data stream without further processing. In the screen-shot on the left-hand side of the "Layer 7" window, the commands in the order of their arrival are listed, whereas in the right pane the details of a chosen command are printed. The raw view of the APDU is shown in the "Layer 2" window below.

2.2.2 Manual Implementation

Within the framework of a product, established ten years ago at COMPRION, the concept for a protocol monitoring implementation has been designed. Providing a solid basis, this design remained unaltered up to the present, and was used in multiple succeeding products. It employs components called *translators* and *analyzers*, which embody the manually implemented data dissectors. Although both component types are designed to work independently, they exhibit a similar parsing front-end. The former use it for greedily printing all encountered structural elements, while the latter focus on specific data fields and the sequence of commands, matching those against stated requirements. They are involved in automated standard compliance tests.

Equipped with a current copy of a specification document, a software developer at COMPRION has to transcribe structure declarations, order restrictions and sev-

eral informally described constraints, to operational code, that accurately matches the semantics of the standard. Using C++ the developer has to ensure correct memory (de)allocation, implement a lot of boundary checks and error handling, accumulating formatted information for the output, by means of concepts available in C++. Then the code can be integrated into the monitoring application, offering the necessary input and output interfaces.

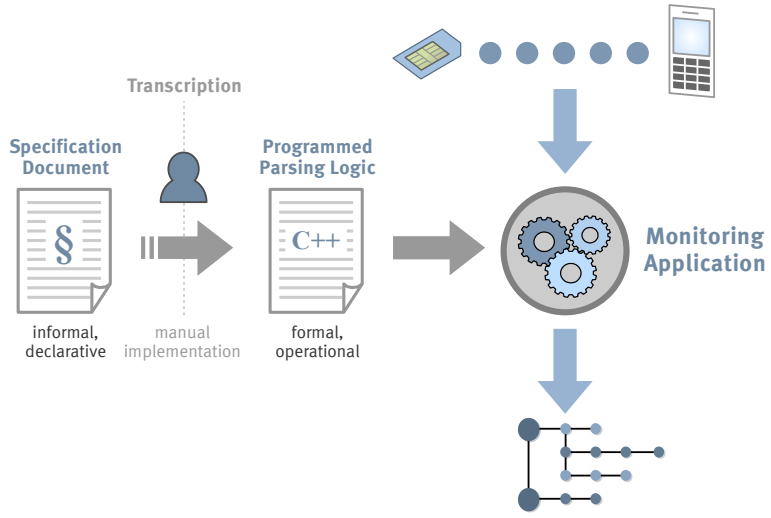


Figure 2.5: *The software developer transcribes aspects from the declarative representation of the standard to concrete operational code. Implicit or informally described constraints have to be formalized and implemented also.*

Nowadays the set of supported technologies includes numerous communication protocols serving a lot of smart card applications. Hence, the amount of realized *translators* and *analyzers* to accomplish this, is quite extensive and still growing steadily. However, the manual implementation approach suffers some critical deficiencies.

2.2.3 Drawbacks

Upon the decision to introduce a new product, with improved hardware along with optimized monitoring applications based on Microsoft's .NET platform, the drawbacks of the current solution became noticeable. The most outstanding are assembled in the following list.

- **Large semantic gap**
  During transcription the developer not only has to be familiar with the context of the specification, but also has to rewrite declarative text as imperative code, while formalizing implicitly assumed or informal descriptions. Spanning this semantic gap requires experience in both domains and is very likely to be a source of errors. It also represents a high entry threshold for developers new to the task.

- **Hard to maintain**
  Any code updates enforced by changes in standards, result in cumbersome reverse engineering of C++ code to allow modification.
- **Re-invention of ever recurring concepts**
  Frequently recurring implementation of boundary checks, value comparisons and case handling, vary in constants and combination. However, the general structure of these problems is not targeted. Also, the heavy use of bit- and pointer arithmetic, which is prone to errors, lacks of a systematic aid. This threatens robustness and therefore increases testing and bug-fixing effort.
- **Re-implementation too costly**
  A comparable reimplementation using another programming language would be too costly, consuming a lot of time. Moreover, the result would exhibit the same strong language binding, again restraining further evolution of products.
- **Lack of portability**
  The close programming language dependency, combined with the large amount of existing parsing routines makes it unfeasible to reuse the hard-coded semantics on the new platform.
- **Inefficient architecture**
  Translators and *analyzers* perform the same parsing twice, wasting resources and increasing the number of potential errors unnecessarily.

Summarizing, we may assert that the development strategy of standard compliant binary parsers, from specification documents is not sufficiently addressed by the current solution. A switch in the representation paradigm, the complex details of both domains and the large amount of similar parsers to handle, make manual implementation an inadequate approach. In order to address these issues, we aimed the introduction of a domain-specific language, which could bridge at least one part of the semantic gap, that is, mapping code from the declarative to the operational representation. While the rest of the gap - the specification formalization - would still be performed by the software developer, the representation switch can be shifted to a compiler.

2.3 Aspects of Compiler Theory

In this thesis we present PBPG and its language, as a linguistic approach to construction of binary parsers. The language design, the development of the compiler and the binary parsers it generates rely heavily on techniques of compiler construction. Hence, we introduce in this section basic notions of compiler theory, including lexical analysis and the analysis of syntax and static semantics. We mention the theoretical models and automata used in association with regular expressions and deterministic context free languages. Thereafter we discuss aspects of semantic analysis and conclude the section by introducing the basic notions of synthesis. Throughout this section we refer to a few books [ALSU06, CT03, Kas90, Wir05], which provided the most information cited. Other sources provided a few additional information.

2.3.1 Compilation is Translation

Communication is always based on at least one language. The form and complexity of the language is influenced by the context of its use and the characteristics of the communication participants. Languages for machine-machine communication differ from languages for man-machine interaction. The former typically allow a bidirectional data exchange on a rather even abstraction level. The latter enables humans to express intentions in terms of a textual or graphical *source language*, which can be transformed to a representation suited to the machine, the *target language*. In this case, where different abstraction levels have to be bridged, compilers can be employed to translate the meaning from the source to the target language [Wir05].

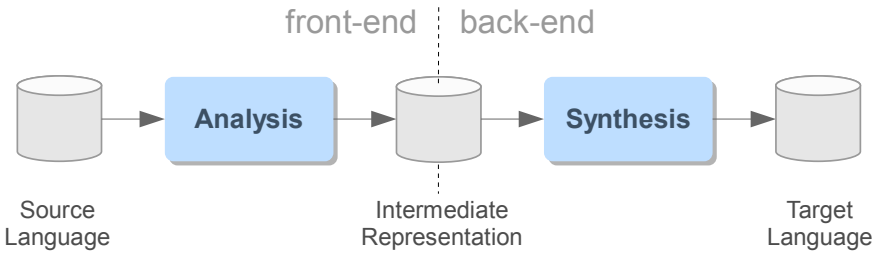


Figure 2.6: *The compilation task is subdivided in an analyzing and a synthesizing part.*

In the construction of compilers characteristics of both the source and the target languages are addressed. The analytic part of the compiler, which processes the source language input, encapsulates the *front-end*. The synthesizing compiler part in contrast, which generates the target language code, represents the *back-end*. In between, an *intermediate representation* constitutes a weak link between the two ends. This division has practical significance. Making both parts exchangeable, it facilitates the portability of the compiler.

2.3.2 Analysis

The function of the compiler front-end is accomplished by a series of analysis phases with increasing abstraction levels. For each of them standard processing techniques are available. We will discuss them one after the other.

Starting with the *lexical analysis* of the input data, the *Scanner* recognizes cohesive character sequences as symbols of the language and classifies them in terms relevant for the later syntax analysis. Constantly discarding white-spaces, line-breaks and irrelevant delimiters, the symbolic stream is transformed to a token-sequence for the parser.

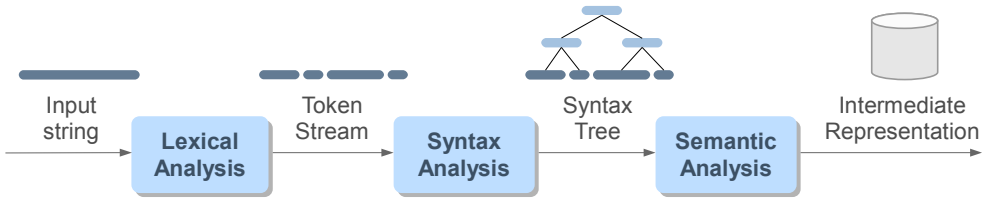


Figure 2.7: *The different steps of analysis focus on symbol patterns, the syntax and the semantics of the input string.*

During the *syntax analysis*, the structure of the token-sequence is matched against the specified language grammar, and made explicit by incrementally assembling elements of the *syntax tree*. In case of successful processing, the input data is proved to be syntactically well-formed.

Based on the syntax tree, the analysis of *static semantics* can be done. It encloses the analysis of identifier dependency relations, type-compatibility, expression evaluation and other compliance with specified rules of the source language.

2.3.3 Lexical Analysis

While the analysis of syntax and static semantics is done by the parser, the lexical analysis is hidden in the scanner. This decoupling of the scanning task is justified by mainly three reasons [ALSU06, CT03]:

- Syntactical analysis can be considerably simplified if the parser does not have to handle white-spaces, line breaks and comments, but can specialize on the syntactical structure of the relevant words only. This purifies the grammar specification and eases the development process of parsers.
- Applying techniques specialized to lexical analysis solely, along with efficient input buffering methods, can improve the overall analysis performance.
- Compilers of related or similar programming languages might share a single scanner implementation while encapsulating their differences in the parser. On the other hand, there exist languages where the application of distinct scanners for different regions of the input document simplifies the overall scanning process.

The construction of scanners can be automated to a great extend, requiring just a little developer intervention. Provided a lexical specification of the source language, a *scanner generator* creates an efficient implementation of the scanner for that language. Hence, the major effort in scanner construction is the correct formulation of the lexical specification.

The core of generated scanners constitutes an Finite State Automaton (FSA), which is configured to recognize and classify character sequences according to the specification. The patterns to be matched are formulated by means of *regular expressions* or other equivalent foundations that FSA can be generated from [Kas90]. The process and the algorithms leading to the final FSA implementation can be found in [CT03, ALSU06].

Regular Expressions

For the specification of character patterns regular expressions can be used. They are constructed by concatenation, alternation or repetition of sub-expressions. Assuming that Σ is the set of available symbols, and $L(X)$ and $L(Y)$ are the languages defined by the regular expressions X and Y, then the table 2.8 specifies the Language $L(R)$ of the regular expressions R.

R	$L(R)$	Description
ϵ	$\{\epsilon\}$	The set containing the empty word
a	$\{a\}$	The set containing $a \in \Sigma$
$X\|Y$	$L(X) \cup L(Y)$	Alternation
XY	$\{xy\|x \in L(X), y \in L(Y)\}$	Concatenation
X^*	$\{\epsilon\} \cup \{xy\|x \in L(X^*), y \in L(X)\}$	Repetition (Kleene closure)
(X)	X	Grouping

Table 2.8: *Languages $L(R)$ defined by Regular Expressions R, in accordance to [Kas90]. The latter four expressions are sorted in the order of their precedence.*

We appoint the alternation-, concatenation- and repetition operators to be left-associative and their precedence to grow in that very order, being lowest for the alternation operator. Using that definition we now can write patterns like the following one, forming the language of hexadecimal numbers with the '0x' prefix.

$$\begin{aligned}\mathrm{HexNr} = 0\mathrm{x}(&0|1|2|3|4|5|7|8|9|\mathrm{a}|\mathrm{b}|\mathrm{c}|\mathrm{d}|\mathrm{e}|\mathrm{f}|\mathrm{A}|\mathrm{B}|\mathrm{C}|\mathrm{D}|\mathrm{E}|\mathrm{F}) \\ &(0|1|2|3|4|5|7|8|9|\mathrm{a}|\mathrm{b}|\mathrm{c}|\mathrm{d}|\mathrm{e}|\mathrm{f}|\mathrm{A}|\mathrm{B}|\mathrm{C}|\mathrm{D}|\mathrm{E}|\mathrm{F})* \qquad (1)\end{aligned}$$

Since this looks somewhat awkward, we may want to add some *syntactic sugar* to the upper definition, which allows a more compact notation. First we can abbreviate HexNr by introducing the non-empty repetition $X^+ = XX^*$. This type of repetition is used very often, and shall therefore get its own operator. Another operator $X? = X^0 \cup X^1$ in this context restricts the Kleene repetition to one occurrence at most.

To abbreviate long enumerations of symbol alternatives, as required for the aggregation of *character classes*, we can use the bracket notation. Further shortening can be achieved by exploiting the encoding order of consecutive symbols, writing only the bounds of a range separated by a hyphen.

$$\mathrm{HexNr} = 0\mathrm{x}[012345789\mathrm{abcdefABCDEF}]^{+} \tag{2}$$

$$\mathrm{HexNr} = 0\mathrm{x}[0-9\mathrm{a}-\mathrm{fA}-\mathrm{F}]^{+} \tag{3}$$

Sometimes it is more convenient to specify character sets using exclusion. In those cases, a leading caret symbol within a character class definition denotes a complementary set, e.g. the set of characters not contained in the pattern.

$$\mathrm{TextLine} = [\hat{}\backslash\mathrm{n}]^{*}[\backslash\mathrm{n}] \tag{4}$$

The language of $\mathrm{TextLine}$ consists of words terminating with a line-break, with all of the prior symbols being anything but a line-break. This is denoted by the escape symbol $\backslash$ followed by n. The same escaping convention applies to the use of all symbols, which are their selves part of the regular expression notation, i.e. $(,), [,], ^{*}, ^{+}, \backslash$ and ^. Further mathematical properties of regular expressions can found in [ALSU06].

2.3.4 Syntax Analysis

The second phase of the analysis concentrates on a higher-level structure of the input. Whereas lexical analysis performs on *word*-level, the syntax analysis operates on the *sentence*-level. Its primary task is the verification of the token stream from the scanner, with regard to the language grammar. The results can take on two forms, either the input complies with the grammar, or not, which makes parsing a classification task, consisting of two related aspects. The formal definition of a language grammar, and the corresponding parsing methods.

Context-Free Grammar

From the universe of possible languages, the class of *context free languages* has the most importance to language design in computer science. This is due to the characteristic capability of Context-Free Grammars (CFGs) of creating sufficiently expressive (complex) syntax for programming languages, which are yet simple enough to allow efficient parsing. That is in time n^3 for an input token sequence on length n [ALSU06]. The set of context-free languages encloses the subset of regular expressions, advanced by recursion beyond the definition of the Kleene repetition [Wir05], as syntactical expressive means. This extension, however, requires a more sophisticated model and techniques for the analysis process, which can not be accomplished by state machines anymore. After introducing the CFG formally, we will present the two standard techniques of *top-down* and *bottom-up* parsing, which cover the subsets $LL(k)$ and $LR(k)$ of context-free languages. Both of them rely on the *push-down automaton* model.

A CFG is a set of generating rewrite rules (*productions*), which specify the construction of sentences of a language from symbols, forming its syntax. In accordance to [Kas90] we use the following definition.

Definition 1 *A context-free grammar G is determined by a four-tuple $G = (T, N, S, P)$, where T is the set of terminals, N the set of nonterminals, $S \in N$ the start symbol, and $P \subseteq N \times V^*$ the set of productions, where $V = T \cup N$ is the vocabulary of G.*

Productions are denoted using $\rightarrow$. For a sequence $w = uAv$ of symbols in V and the production $p \in P : A \rightarrow x$, the application of p on w (*derivation*) is written as $uAv \Rightarrow uxv$. The incremental derivation using recursive productions, such as $A \rightarrow Aa$, is written using $\Rightarrow^*$, like in $A \Rightarrow^* aaa$. It means that the sentence on the right-hand side is *derivable* from the nonterminal on the left. The set of sentences derivable from the start symbol of a grammar G is the language $L(G) = \{w | w \in T^* \wedge S \Rightarrow^* w\}$ of the grammar. The restriction to a single nonterminal on left-hand side of productions in P makes the grammar and the language it generates *context-free* [SK95]. Grammars without this restriction are *context-sensitive*.

CFG are typically written using the Backus-Naur Form (BNF) or Extended BNF (EBNF) as an abbreviating notation. The former allows collections of productions with the same nonterminal on the left-hand side to be written as one production with multiple alternatives on the right-hand side. Further shortening is achieved by using EBNF, which advances BNF by arbitrarily nested regular expressions on the right-hand side. We summarize these shortcuts in table 2.9.

EBNF	EBNF alternative	equivalent BNF with $X \rightarrow uYw$ and	Description
$X \rightarrow u(v)w$		$Y \rightarrow v$	grouping
$X \rightarrow u[v]w$	$X \rightarrow uv?w$	$Y \rightarrow v \vert \epsilon$	optional occurrence
$X \rightarrow us^*w$	$X \rightarrow u\{s\}w$	$Y \rightarrow sY \vert \epsilon$	optional repetition
$X \rightarrow us^+w$	$X \rightarrow us...w$	$Y \rightarrow sY \vert s$	repetition
$X \rightarrow u(v \Vert s)w$		$Y \rightarrow vsY \vert v$	delimited repetition
$X \rightarrow u(v_1 \vert v_2)w$		$Y \rightarrow v_1 \vert v_2$	alternative

Table 2.9: *EBNF abbreviations for equivalent BNF production pairs, where $X, Y \in N$ $u, v, v_1, v_2, w \in V^*$ and $s \in V$, in accordance to [Kas09a]*

In figure 2.8 we show a simple grammar G_{lr}which defines the language $L_{lr} = L(G_{lr}) = \{l^n r^n | n \in \mathbb{N}^+\}$. Its sentences consist of an equal number of `l` and `r` terminals. Starting with p_1 the start symbol *S* is replaced by the nonterminal *M*. Then each derivation using p_2 replaces *M* by itself with enclosing `l` and `r` terminals. The incremental derivation adds further terminals within the expression from prior iterations. A final application of p_3 stops the derivation process, yielding a sentence in L_{lr}. This derivation can be drawn as a tree, the derivation tree or *syntax tree*.

$G_{lr} = \{T, N, S, P\}$
$T = \{l, r\}$
$N = \{S, M\}$
$P = \{p_1, p_2, p_3\}$

p1: $S \rightarrow M$
p2: $M \rightarrow \texttt{l}\ M\ \texttt{r}$
p3: $M \rightarrow \texttt{lr}$

Figure 2.8: *Grammar G_{lr} for the language $L_{lr} = \{l^n r^n | n \in \mathbb{N}^+\}$.*

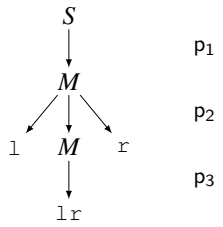


Figure 2.9: *Derivation tree for the sentence $llrr \in L_{lr}$.*

This example illustrates that a context free language can be defined, which can not be handled by an FSA, i.e. which is not regular. Although it is describable in terms of BNF notation $\{l^i r^j | i, j \in \mathbb{N}^+\}$, using an FSA we have no means to guarantee a balanced number ($i = j$) of both terminals, since FSA cannot count [ALSU06].

Concrete vs. Abstract Grammars

For a language two types of grammars can be given, an *abstract* and a *concrete* grammar. A concrete grammar is unambiguous and specifies delimiters and other terminals, to support the syntactical structure. It defines the concrete syntax of the language from the perspective of a programmer. An abstract grammar on the other hand contains condensed information only, omitting all symbols of no value for the further analysis. It defines the abstract syntax of the language, which is ambiguous in most cases. Please consider the following ambiguous expression grammar. Its sentences are arithmetic expressions written in infix notation.

p1: *Expr* → *Expr Opr Expr*
p2: | `ident` | `number`
p3: *Opr* → `+` | `-` | × | ÷

In this grammar the same arithmetic expression $a - b + c$ can be constructed by leftmost, and rightmost derivation of *Expr*.

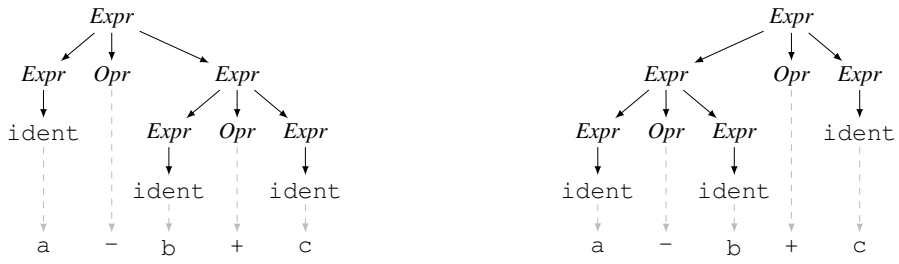


Figure 2.10: *Leftmost and rightmost derivation of the same input $a - b + c$.*

The tree on the left of figure 2.10 groups the input as being right-associative i.e. $a - (b + c)$, whereas the tree on the right groups the identifiers in a left-associative way $(a - b) + c$, which is semantically unequal. For the concrete syntax tree, derivations

must be performed unambiguously. Thus, the expression grammar has to be specified with more structural information.

p_1:	*Expr*	→	*Expr AddOpr Term*	p_5:	*Prod*	→	`ident`
p_2:		|	*Term*	p_6:		|	`number`
p_3:	*Term*	→	*Term MulOpr Prod*	p_7:	*AddOpr*	→	+ | −
p_4:		|	*Prod*	p_8:	*MulOpr*	→	× | ÷

The previous right-recursion of *Expr* is eliminated. Instead, two nonterminals *Term* and *Prod* are introduced, and the terminals of *Opr* are split into distinct nonterminals *AddOpr* and *MulOpr*. The left-recursive definition of *Expr* and *Term* makes the addition and multiplication operators left-associative. The stepwise operator definitions introduce an order of precedence, which increases with the depth in a chain of productions [Kas09a]. Hence, in this grammar multiplication has higher precedence than addition. This way schematically constructed grammars disambiguate the derivation process imprinting associativity and precedence in the concrete syntax.

Parsing Techniques

As aforementioned, parsing CFG is harder than parsing regular expressions. Hence, a more powerful model has to be employed to do this task. In particular the Pushdown automaton (PDA) model has exactly the desired quality. It is an FSA advanced by a stack, whose top element serves as additional input to the transition function, represented by a table. Parsing an input string is now considered as stepwise derivation of

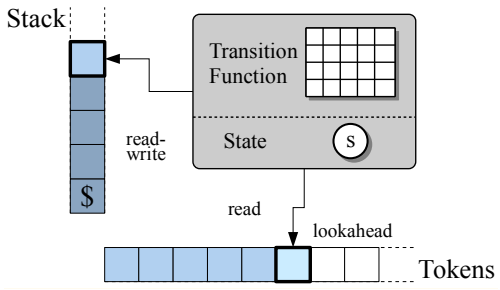


Figure 2.11: *A parser simulates basically a PDA. It employs a stack, whose top element serves as additional input for the transition function.*

the input, depending on the current PDA-state, an input symbol and the top-element of the stack. Accordingly, successful parsing is not reached by dedicated automaton states solely, but requires the entire input to be read - indicated by $, and the start symbol as the final symbol on the stack above $.

There exist two standard ways for parsing context-free languages, *top-down* and *bottom-up*, which concerns the way of constructing the derivation tree. Top-down parsing builds the tree from the root to the leafs in prefix order, by applying productions forwards (as they are defined). In contrast, bottom-up parsing assembles the tree

from the leafs upwards, node by node towards the root, by inverse application of productions, i.e. by *reduction*. For a sequence of derivations $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n = \alpha$ reproducing the input string α, the bottom-up approach calculates $w_{n-1} \Rightarrow w_n$ before it calculates $w_{n-2} \Rightarrow w_{n-1}$. This difference in the direction leads to distinct classes of languages, supported by top-down and bottom-up parsers.

Bottom-up parsers cover the set of $LR(k)$ languages, which is the set of deterministic (or unambiguous) context free languages, while top-down parsers support just a subset $LL(k) \subset LR(k)$ of languages [Kas90]. In both cases, the first L denotes the input processing direction being left-to-right, whereas the second L or R stand for *leftmost derivation* and *rightmost derivation* accordingly. The argument k represents the lookahead boundary. It is the number of tokens beyond the current token position the parser may peek at in order to deterministically identify the next production. Both alternatives perform in time $O(n)$ for an input string of length n [ALSU06]. Fortunately for typical programming languages $LR(k)$ or $LL(k)$ grammars with $k = 1$ can be designed, offering efficient compilers for them.

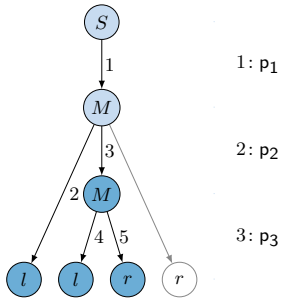


Figure 2.12: *Syntax tree after top-down parsing of the input $llrr$ up to the first r.*

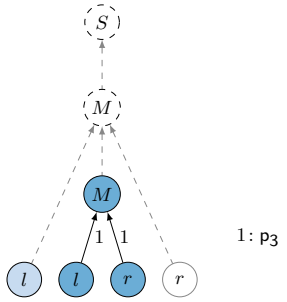
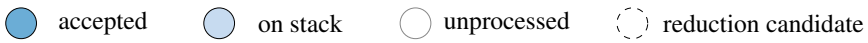


Figure 2.13: *Syntax tree as constructed by a bottom-up parser for the same input $llrr$.*



Top-Down Parsing

Following the natural structure of a CFG, top-down parsing is a rather intuitive working principle, and the only viable way to program parsers manually. From the start-symbol on, a top-down parser creates the root and the nodes of the derivation tree in a depth-first order. By application of leftmost derivations, it grows the tree towards the leafs. This is illustrated in figure 2.12. In this situation the nonterminals *S* and *M* of p_1 are partially derived (on stack), whereas the terminal l and the second *M* have already been accepted by means of p_3, during production p_2. By reading the remaining symbol and successfully matching it against the last terminal of p_2, the input will be accepted, i.e. *M* and *S* will be removed from the stack, completing the tree construction.

One way to implement a top-down parser is the *recursive descent* approach. It is based on mutually recursive parse functions, each responsible for parsing of one certain

nonterminal of the grammar. The dependencies of nonterminals are reflected in the structure of function calls in the implementation, where occurrences of nonterminals on the right-hand side of productions are mapped to function-calls. Every function returns a Boolean value indicating the parsing success, which is given if the function successfully consumes the input according to the definition of the nonterminal. The overall process is successful if the function of the start symbol finally returns true. This is shown in the parsing code in figure 2.14 for our language L_{lr}.

```
1:  global token
2:  function AcceptS
3:      token ← getNextToken()
4:      return AcceptM()
5:  function AcceptM
6:      if token ≠ l then
7:          return false
8:      token ← getNextToken()
9:      AcceptM();
10:     if token ≠ r then
11:         return false
12:     token ← getNextToken()
13:     return false
```

Figure 2.14: *Exemplary recursive descend parser for the language L_{lr}.*

In the simplest case of recursive descent, parsing of valid input alternatives is done by probing all corresponding functions in an exhaustive trial-and-error manner. In case of a mismatch, the process has to backtrack to the most recent previous valid state, and to try other ways to match the input. If the last opportunity fails, the parser must stop with a meaningful error message.

The amount of performed backtracking before choosing the right production obviously affects the runtime of the algorithm. Probing alternative ways to match the input blindly, i.e. based on passed tokens only, is therefore not very efficient. In many cases the knowledge of one or more tokens beyond the current one (*lookahead*) unambiguously determines the correct production to choose next, i.e. the correct parse function to execute. This can push the number of backtracking to zero, defining deterministically correct steps for each new input symbol.

An implementation of a recursive descent parser can either be direct-coded, as indicated in figure 2.14, or it can be table-driven. The former type utilizes the runtime stack of the executing machine as its pushdown automaton to handle parsing states and perform backtracking implicitly. Such parsers typically perform very fast and are chosen in case of manual parser implementations. They have the advantage of a simple architecture, combined with good error detection and correction capabilities [CT03] along with all the freedom of manual implementation.

The table-driven approach on the other hand, employs an explicit push-down automaton driven by a lookup-table based transition function. Such parsers are typically generated by parser construction tools, and perform slower. However, generated parsers benefit from smaller implementation effort high-level specifications, which reduce the development costs and increases robustness.

Bottom-Up Parsing

As indicated in figure 2.13, bottom-up parsing creates the syntax tree from the leafs towards the root in postfix order. While the input is still processed from left to right, the application of productions is reversed. This inverse derivation is called *reduction* and means the replacement of totally accepted production bodies by their left-hand sides. The final goal is the reduction of the entire input string to the single start-state (or goal-state), the root of the derivation tree. Bottom-up parsing allows to use the more powerful $LR(k)$ grammars, which overcome the limitations associated with $LL(k)$ grammars, e.g. left-recursion and alternative productions with equal beginnings. However, due to the complexity of the concrete parsing procedure, the construction of LR parsers can not be performed by hand practically, but is rather performed by construction tools.

In order to enlarge the set of supported grammars, to overcome conflict-situations, a lookahead of $k > 0$ is required. Additionally, improvements on the parsing techniques can be done. Today $LALR(1)$ represents the most powerful but still efficient parser class. It is supported by many parser generators being a standard in compiler construction. Detailed information about properties and construction of bottom-up parsers can be found in [Kas90, ALSU06, CT03].

2.3.5 Semantic Analysis

The pure recognition of the input syntax is not enough. Beyond the syntactical structure there is semantic information an input sentence transports, which has to be checked. Aspects of the semantic analysis are primarily *name analysis* with respect to the implemented scope rules and *type analysis* involving checks of type relations and consistency. Depending on the language and its application domain, semantic analysis is performed at compile-time (for statically typed languages) or at run-time. The validation of program's compliance with semantic rules and constraints of a language is the last task of the compilers analysis front-end.

The basis for this analysis is the abstract program tree, upon which tree-walking algorithms calculate values, associated to semantic properties or *attributes* of grammar symbols. Depending on the size and complexity of a language and its compiler, semantic analysis may involve automated or manual techniques.

A formal and powerful technique are Attribute Grammar (AG), introduced by Donald E. Knuth in [Knu68]. This method defines a declarative annotation of CFG, to define semantic relations and constraints between attributes of grammar symbols. From

the dependencies defined by the attribute relations, the required order of value propagations and computations, forming a tree-walk plan, can be derived automatically.

Alternatively tree-walks can be implemented manually as well. Also, the choice of the employed parser generator might allow manual implementation only, lacking support for AG. Regardless of the chosen approach, attribute evaluation requires a good understanding of the the problem and its complexity though. An overview of the algorithms and further explanations on the systematic construction of attribute evaluators for AG can be found in [Kas90].

Syntax-directed Translation

Manually implemented semantic analysis requires the prior construction on the abstract program tree, which is based on the concept of *syntax-directed translation* [CT03]. This technique became popular with the advent of the parser generator *yacc* (yet another compiler compiler) by S.C. Johnson in the 1970s [LJ09]. Similar to AG it involves augmentation of grammar productions. However, it is less formal, in that it accepts arbitrary user-defined code, referred to as *semantic actions*. These *semantic actions* must be provided in the syntax of the language, the parser is constructed in, and are executed at run-time of the parser, whenever associated productions are applied. Here we see the grammar for the language $(a(,a)^*;)$, annotated with semantic actions to accumulate occurrences of `a` into a list object:

```
p1:  S  →  A ';'        { $$ = $1; }
p2:  A  →  A ',' a      { $$ = $1; $$.Add($3); }
p3:     |  a            { $$ = CreateList(); $$.Add($1); }
```

Assuming that this grammar is designed for a bottom-up parser, the semantic actions are triggered in the same way, and so shall be read. In p_3 we see the instantiation of the list object using `CreateList()` upon the first occurrence of `a`, and its assignment to $$. Then `a` is added to the list using `Add($1)`. Whenever p_2 is applied, for each further occurrence of the string `',a'` the list object is propagated from $1 to $$ with an `a` added. As soon as `';'` causes reduction to the start state, the list object is again assigned to $$. The symbols $$ and $1, $2, ... represent local references to objects, associated to the symbols of the production. $$ references the object associated with the nonterminal on the left-hand side of the production, whereas $1, $2 and $3 point to objects of symbols in the production body. The code inside the semantic actions braces $\{\dots\}$ must be valid in terms of the host language, of the parser implementation, where it is blindly copied into, by the generator. In our case, this might be an object-oriented language like C++. Properties and methods of objects programmed in the context of the host language, represent attributes of associated symbols in the context of the grammar. Thus, assembly of the abstract program tree, using host language functionality, directed by the derivation process, yields a hierarchy of object instances reflecting abstracted and attributed structure of the input.

Tree walking

Analysis of static semantics requires the program tree as input. This can be done using mutually recursive validation methods, implemented for each kind of tree node. Then the validation of the tree is initiated from its root, and performed until all subroutines return successfully or raise an error. However, care must be taken to ensure termination of the whole procedure, avoiding recursion loops.

Depending on the complexity of the rule, the required effort may vary, incorporating propagation of values top-down and bottom-up. For instance, in order to check the constraint "defining occurrence before applied occurrences" (def-before-use), as implemented in C (*C scope rules*), one tree-pass suffices. However, if the *Algol scope rule* is to be checked, which allows applied occurrences prior to a defining occurrence, within a scope, two tree passes must be done to prove compliance [Kas90]. A first pass for the registration of defining occurrences, and the second for the validation of applied occurrences. We illustrate the situation in the following figure.

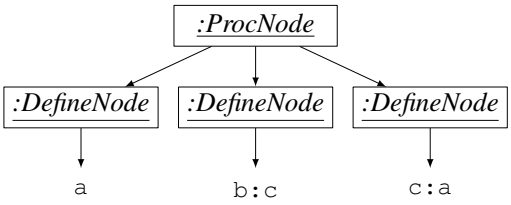


Figure 2.15: *Abstract program tree for a procedure body with three definitions.*

Assuming that the horizontal order of the *DefineNode* instances reflects the textual order from the input code, the definition of `a`, `b` and `c` would be accepted as valid, in terms of the Algol scope rules. However, in terms of C scope rules the applied occurrence of `c` during definition of `b` would evaluate to a rule contradiction.

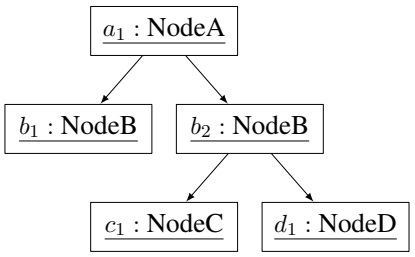
Validation of typing rules, the correct operator-handling and evaluation of expressions can be performed by the same tree-walking algorithms. A good strategy is to gather as much information as required during construction of program tree nodes, and to keep the amount of distinct node types as small as possible, to reduce the effort during tree-walks. Furthermore, use of dedicated property tables describing types, structures or values makes a clean parser design, to separate distinct concerns.

2.3.6 Synthesis

As soon as all analysis steps are passed successfully, and the abstract program tree is created and attributed, the constructive part of compilation may be initiated. Tuning the intermediate representation of the input, its alignment towards the target system, and the code generation, are the major tasks of this phase. For low-level, hardware-related target languages synthesis demands a thorough study of the target machine's specification. Its memory management, the set of supported instructions, and characteristic features or restrictions have major impact on the target code synthesized, and the effort required to implement the back-end of the compiler.

However, in our project the target code belongs to the high level language C#, which abstracts from hardware-specific machine characteristics, and which can be relied on to take over a lot of optimization effort. Nevertheless, the target structures, which our intermediate representation must be mapped to, have to be defined. Also, the context of the target code execution, including the involved interfaces and data types must be taken into account.

Disregarding the optimization of the abstract program tree, or the intermediate code representations, the process of code generation is straight forward. Similar to the tree-walk based attribute evaluation, the target code can be emitted in a recursive traversal of the tree.



1: a_1*_prologue*
2: b_1*_prologue (*b_1*_epilogue)*
3: b_2*_prologue*
4: c_1*_prologue (*c_1*_epilogue)*
5: d_1*_prologue (*d_1*_epilogue)*
6: b_2*_epilogue*
7: a_1*_epilogue*

Figure 2.16: *Structured output code as might be generated by traversing the program tree on the left. A prefix visit of a node creates associated* prologue code*, a postfix visit* epilogue code *accordingly.*

The visited nodes emit predefined code templates parameterized with dynamic content, which has been gathered during analysis. In order to create enclosing structures, such as $\{\dots\}$ in C, or `begin-end` in Pascal, emission of appropriate code-chunks must be done during preorder and postorder visits. The overall generated structured output may then be embedded into an execution framework, connected to the surrounding environment.

The synthesis phase is discussed in much more detail in several books including those we referred to throughout this section [ALSU06, CT03, Kas90]. We conclude the section of traditional compilation methods, footing on processing textual input, and focus on the less explored field of parsing binary coded data.

2.4 Parsing Binary Data

In clause 2.3.2 we argued that the decoupling of the analysis task into *scanning*, *parsing* and *semantic analysis* is well-grounded, being motivated by benefits in efficiency, clarity of design, maintenance and extendability. For textual language processing this cascaded analysis is known to work very well. We show the front-end architecture from page 16 once more in figure 2.17.

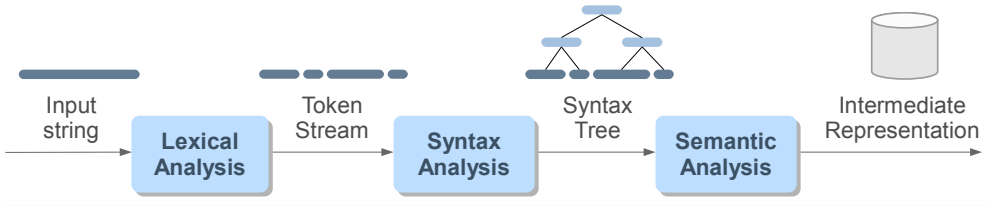


Figure 2.17: *Architecture of the analysis front-end.*

In this section we raise the question and try to argue why binary data cannot be parsed the same way using the same procedures. Therefore, we expose the characteristic properties of the two types of input and their differences. This will lead to the conclusion of properties of a binary data parser and its construction.

- **Granularity and Encoding of Symbols**
  Textual representation as expected by a traditional lexical analyzer builds on characters. Depending on the standard used, a single character consumes one byte (*ASCII*) or multiples of a byte (*Unicode*) of memory. In any case, a homogeneous immutable encoding scheme is imposed on the input data. In contrast, the granularity of binary data is not restricted in such a global way. Data structures of arbitrary formats for arbitrary purposes, enclosing communication, often define bit-size grained data fields, sometimes spread over byte boundaries (*bit packing*).

- **Explicit vs. Implied Structure**
  In clause 2.3.3 we introduced the notion of character patterns and character classes formulated by regular expressions. All textual programming languages have in common that they take a few major character classes for the construction of composite patterns: *letters*, *numbers*, *symbols* and *white spaces*. The latter two classes have a delimiting property required for localization of tokens and their boundaries. Since there is no such commonality among binary data representations, the absence of dedicated delimiters causes a lack of explicit structure in the input. The usual notion of a token looses significance. This leads to the necessity to make implications on structure and values based on the context. As a result, binary data structures not only can be considered as non-regular but even non-context-free.

With these arguments in mind, we may conclude that despite the seemingly marginal modification of the type of input, the impact on the architecture of the analysis front-end might be greater than expected. In particular, the new problem faced suggests the removal of the boundary between the scanner and the parser, so far justified by the advantage of tokenization. As a result, the new merged analysis step - we might call *dissection* is directly coupled to the semantic analysis. Furthermore, the decision on how to proceed parsing from a certain situation depends on evaluation of gathered context information. This demands the semantic analysis to control the dissection process, creating a loop between dissection of new data and its semantic evaluation. With these modifications applied we obtain a system shown in figure 2.18, which reflects a first model of a binary parser.

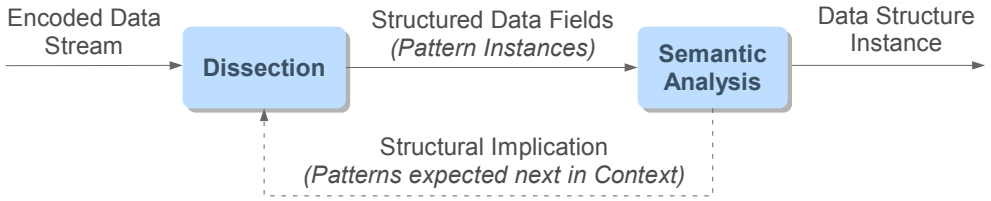


Figure 2.18: *Architecture of analysis steps as required for binary data structures.*

The important consequence of this circular dependency is that in the general case[5], the data beyond the current analysis position cannot be accessed directly, i.e. it cannot be dissected (or even located) without dissection of all the preceding data.

At this point doubts might emerge that context dependency could impede an efficient function of this model. For a large scale of binary encoded languages this might probably be the case. However, with regard to the intention of programmers, to use binary data structures as means to store or transport information between computer systems in an efficient way, we may expect that reasonable structure specifications avoid usage of hard-to-decode structures. In the special case of smart card communication, the protocol specifications are designed for implementation in comparably simple devices with very limited resources. Hence, a set of simple structural patterns in combination with *indication* techniques are used to support fast data processing.

2.4.1 Structural Patterns and Indication

Many programming languages offer syntactical elements to express *sequences*, *alternatives* and *repetitions* as data types. They are used to create composite structures by nesting and are often called *structs*, *unions* and *arrays*. In this context binary data structures can be imagined as serialized versions of such composite types. Actually, this is how binary data structures are often treated. Being modeled in terms of *structs*, *unions* and *arrays*, deserialization is performed by blindly reading in a byte-stream of

[5]For example, a sequence of variable-length data fields with local length indications.

the size of the composite data structure into an instance of it. However, this works for structures of constant size only. Furthermore, using only structural description for serialization introduces ambiguity to the serialized data. That is, the same serialized byte stream might be the result of two different in-memory data representations, which cannot be deserialized unambiguously by means of structural descriptions only. Hence, deserialization (or *binary parsing*) requires a more sophisticated form of a type description, in order to face ambiguity.

Please consider the definition of the union type U_1 in C-like syntax shown below. Given corresponding serialized data like 10 03 AB CD EF which fits into both alternatives S_1 and S_2 of the union, deserialization could not unambiguously associate one of the two struct types to the data.

```
union U1 {
  struct S1 {
    byte tag; // 0x10
    byte len;
    byte[] val;
  }
...
...
  struct S2 {
    byte tag; // 0x20
    long val;
  }
} // 2 ≤ overall size ≤ 100
```

Figure 2.19: *Definition of a union data type U_1 with two struct types S_1 and S_2 nested.*

But, with the additional constraint that valid tag field values are 0x10 for S_1 and 0x20 for S_2 the byte stream can be unambiguously associated to S_1.
This method of *tagging* is one of a couple of frequently used indication patterns, to enable and guide deterministic binary parsing. Alongside there are *length-*, *presence-* and *location* indications which are briefly explained in table 2.10.

Pattern	Example	Description
Tagging	10 03 AB CD EF	An identifying field (*tag*) associating a certain structure specification with the subsequent data; here (10)
Length indication	10 03 AB CD EF	A field indicating the length of subsequent length-variable data structure; here 3 bytes of data follow.
Location pointer	10 03 AB CD EF	Absolute or relative position indication of certain data block within a stream. Here AB CD EF might be a memory address.
Presence indication (*flags*)	1 0 0 0 1 0 1 1	A bit-field consisting of flags indicating presence (1) or absence (0) of certain objective.

Table 2.10: *Established indication patterns often used to guide binary parsing.*

These patterns are applied whenever the binary representation may become ambiguous at the time of serialization. Hence, they represent anchor points or switches guiding the parsing process through the data stream in a meaningful way.

As a result, for the construction of binary parsers means must be offered to model structural patterns, indication techniques and constraining semantic expressions.

2.4.2 A Type System for Binary Parsing

The type system proposed in this clause is based on research of groups of people whose projects we will present in the Related Work section. We outline theoretical aspects of a type description system with the goal to establish a consistent notion of binary data type construction and the process of parsing based on this type system.

We start with the definition of the integral data types, commonly known from many programming languages. The value domain of an integral type can be interpreted as an ordered language of *numeric words*, with the maximum value being the upper boundary and 0 assumed as the lower boundary. Hence, the language of a Byte type is $L(Byte) = \{0, 1, .., 255\}$.

Type	Bit-size	max. Value
bit	1	1
2-bit	2	3
..	..	..
7-bit	7	1F
byte	8	FF
short	16	FF FF
int	32	FF FF FF FF
long	64	FF FF FF FF FF FF FF FF

Table 2.11: *Generic data types of the proposed type system.*

On top of these generic types, composite types can be built by application of structural patterns, we already introduced in different variants for regular expressions and the BNF notation. Let X and Y be arbitrary types of the type system, then T is a composite type defined by the structuring operations in table 2.12.

T	$L(T)$	Description
XY	$\{xy \vert x \in L(X), y \in L(Y)\}$	sequence
X/Y	$L(X) \cup L(Y)$	prioritized alternative
(X)	X	grouping
X^*	$\{\epsilon\} \cup \{xy \vert x \in L(X^*), y \in L(X)\}$	optional repetition
X^+	$L(XX^*)$	non-empty repetition
X^n	$\{x \vert x \in L(X)^* \wedge \vert x\vert = n\}$	constant repetition
$X^{n..m}$	$\{x \vert x \in \bigcup_{i=n}^{m} L(X)^i\}$	bounded repetition
$X?$	$L(X^{0..1})$	optional occurrence

Table 2.12: *Operators for the construction of composite types T from types X and Y.*

The language of a composite type T i.e. the set of binary sequences it defines is enclosed by the structural shape of the type. A concrete word of the type's value domain $L(T)$ is an *instance* $\Im(T)$ of the type T. Through the structure of composite type definitions characteristic value domains of versatile forms and sizes (custom data

structures) can be shaped. For example, let A, B, C be byte types, then the type S_1 from figure 2.19 could have the form:

$$S_1 = (A\ B\ C) \text{ with } |C| = \Im(B)$$

The set of byte fields defined by S_1 encloses the words constructible from the composite language:

$$L(S_1) = \{abc | a, b \in L(byte) \wedge c \in L(byte)^b\}$$

For the convenient and modular type construction an additional relation is required, the *inheritance*. We notate inheritance as $X \lhd Y$ when the type X inherits from Y. In order to allow reuse of frequently required structures, inheritance of types along with *refinement* of substructure by member *overloading*, adds further degrees of freedom in modeling new types. Refinement of substructure would impose additional structural constraints on otherwise plain data fields. For instance, an inheriting type $S_1' \lhd S_1$ could refine the subtype C by assigning further substructure $S_2 = (byte\ byte)$ of constant size to it:

$$S_1' = (A\ B\ S_2) \quad \text{with} \quad |S_2| = |byte| + |byte| = 2|byte|$$

The diagram in figure 2.20 illustrates the construction of data structures by incremental specialization of types, narrowing the domain of representable instances, towards concrete data values.

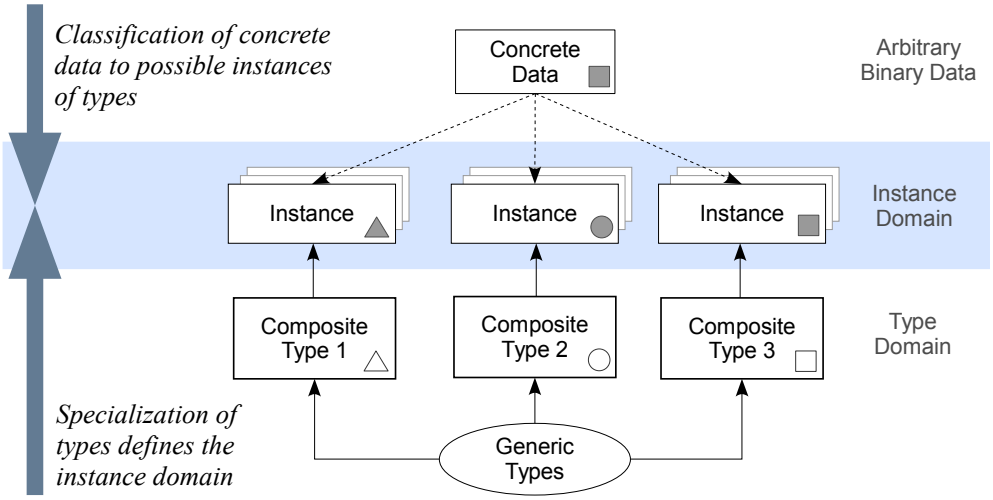


Figure 2.20: *Modeling data structures by incrementally specializing types. The most specialized types shape the form, concrete instances of data may take on.*

Based on this type system, parsing binary data can be best described as a classification task. For a given type hierarchy and an input data sequence, a matching classification of the data to the form and series of expected types is calculated.

Due to the order of the increasing restriction in a hierarchical type definition the evaluation of a structural constraint match is easily performed. For example, if d is a

data instance and $byte_{200} \lhd byte$ with $L(b_{200}) = \{0, .., 200\}$, then if d is an element of the language $L(byte_{200})$ it must also be an element of $L(byte)$, because $byte_{200}$ is more restrictive than $byte$. Thus, in an inheritance chain $T_n \lhd T_{n-1} \lhd \cdots \lhd T_1$, a successful structural match with respect to T_n implies a successful structural match with T_1. However, this holds a particular consequence for the evaluation of matches in alternatives, similar to the situation in figure 2.20. It may happen that d is compared against two or more alternative types, where one type is more general than the other. Then in order to gain most information of the data the comparison must be performed in favor of the more specialized type, leaving the other as a less-prioritized alternative. We emphasized this behavior in the deviated notation of the alternative operator (/ instead of |) in table 2.12.

An additional complication arises with the introduction of semantic constraints, enforcing the evaluation of the entire inheritance chain. This is justified by the lack of order among unrelated semantic constraints. For instance, the constraint C_T of an inherited type $T \lhd Z$ stating *"the elements of* $L(T)$ *are odd numbers only"* is unrelated to the constraint C_Z: *"the values between 10 and 20 are excluded"*. Thus, from the presence of a data instance d in $L(T)$ can not be implied that d is element of $L(Z)$. Hence, to determine if a data instance matches the type T_n including the semantic constraints, all semantic constraints $C_{T_n}, C_{T_{n-1}}, \cdots, C_{T_1}$ of the types of the inheritance chain $T_n \lhd T_{n-1} \lhd \cdots \lhd T_1$ must be evaluated with a positive result.

Finally, we present an initial algorithm to show how binary parsing could actually be done using the type system. Let T be a composite type, C_T its semantic constraint and $\mathrm{Evaluate}(C_T)$ the constraint evaluation function. Then binary parsing is performed in a recursive descent manner as shown in the algorithm in figure 2.21.

Its base principle is to recurse into the specified type-structure and it's substructure in depth-first order towards the elementary (*terminal*) data types. Those consume portions of input data of the corresponding bit-size without further recursing. For each successfully parsed type in the type-structure the associated semantic constraint is evaluated with the potential result to abort further descent, which initiates backtracking.

```
1:  function ParseBinaryType( T ): bool
2:      if (T is a sequence) then
3:          for (subtypes T_i in T_1..T_n = T) do
4:              // recurse
5:              if (false = ParseBinaryType(T_i)) then
6:                  return false;
7:      else if (T is a union of alternatives) then
8:          for (subtypes T_i in T_1..T_n = T) do
9:              // recurse
10:             if (false = ParseBinaryType(T_i)) then
11:                 Remove T_i from T
12:             else
13:                 break
14:         if (T is empty) then
15:             return false
16:     else if (T is a repetition over element type e) then
17:         Let n be the number of repetitions of e in T
18:         for i = 1 to n do
19:             // recurse
20:             if (false = ParseBinaryType(e)) then
21:                 break
22:             Subordinate e to T
23:     else
24:         // native type - the last option
25:         Read |T| Bits from the input into T
26:     end if
27:     // finally evaluate the semantic constraint
28:     Let C_T be the semantic constraint of T
29:     postConditionResult = Evaluate(C_T)
30:     return postConditionResult;
```

Figure 2.21: *The idea of the binary parsing algorithm is to recursively descend into the sub-structure of the type handed in and finally read the required number of bits into the terminal native types.*

2.5 Summary

In this section we outlined the difference between parsing textual and binary input which is the lack of explicit structure in the input and the need to imply structure as indicated by the semantic values in the local context. Parsing techniques from the compiler theory can be applied to the this task, though with a modified analysis model controlled by semantic constraints. We proposed a type system for the specification of data structures in combination with semantic constraints to model frequently recurrent structural patterns and indication techniques. The system is based on elementary numeric types for composite type construction through *sequences*, *alternatives* and *repetition*. Based on this type system we presented a recursive-descent parsing algorithm for binary languages.

3 Related Work

Compared to the familiarity of the Internet as communication technology, the knowledge of smart card communication is predominantly ceded to a few experts, concerned with the development of the associated devices. This reflects in the scientific literature as well. Most of the encountered projects are located in the domain of textual Internet protocols covering *constructive* approaches such as specification, verification and implementation or *analytic* approaches targeting network security issues. Just a few publications address binary data parsing. Making no claim on completeness, we review the most relevant publications for our purpose in chronological order. By the end of the review a summarizing taxonomy of projects and publications will be presented, classified by the kind of the targeted problem. It will also cover works not further discussed in the review.

3.1 PACKETTYPES

We start with a work by McCann and Chandra [JS00] which appeared at Bell Labs in the year 2000, being cited in many latter projects associated with binary data parsing. McCann and Chandra for the first time suggested a type system for convenient expression of binary data structures. Its aim is to relieve cumbersome low-level programming of protocol packet parsing, providing a tool for rapid development of applications for network monitoring, accounting or security services. The language is oriented on the C notation of structs, extending it in several ways. From the bit-level on, the language allows the modular construction of complex type hierarchies of the required granularity. The PacketTypes compiler then generates interfaces and code covering low-level protocol parsing in C as a partial implementation which is integrated in protocol handling applications. As can be seen from the specification of a packet in listing 3.1, the definition is split into a declarative part of member field descriptions, which shapes the set of possible packet instances, and a collection of constraints over the members, narrowing the instance-set.

PacketTypes was a first thrust into type-based specification of binary data structures along with semantics in a formal way. It established the strategy of two-phased modeling: composition and constraining, which exhibits a more powerful expressiveness than composition alone. To support layering and encapsulation of protocols, the language provides operators for *refinement* and *overlaying*. These allow new types to be defined on top of available base structures, with advanced superimposed structure on member fields.

In analogy to the union operator in C, PacketTypes defines an *alternation* operator for fields which may take on a single value among a set of alternatives. In contrast to our considerations the set of alternatives is expected to be disjunctive. This allows instant recognition of the correct choice without probing and backtracking, which on the other hand constraints the set of expressible structures significantly.

```
IP_PDU := {
   nybble version;
   nybble ihl;
   byte tos;
   short totallength;
   short identification;
   bit morefrags;
   bit dontfrag;
   bit unused;
   bit frag_off[13];
   byte ttl;
   byte protocol;
   short cksum;
   ipaddress src;
   ipaddress dest;
   ipoptions options;
   bytestring payload;
} where {
   version#value = 0x04;
   options#numbytes = ihl#value * 4 - 20;
   payload#numbytes = totallength#value - ihl#value * 4;
}

nybble := bit[4];
short := bit[16];
long := bit[32];
ipaddress := byte[4];
ipoptions := bytestring;
```

Listing 3.1: *A packet specification in* PACKETTYPES *is split into a composing and a constraining part [JS00].*

One of the deficiencies of PacketTypes is its fixed set of expressible semantic constraints, including tagging and length-indication aided by arithmetic expressions. Since the PacketTypes language is closed to its host language, frequently required operations such as checksum calculation, packet decryption or custom status reporting have to be triggered from outside the generated code. Furthermore, the generated code was lacking any error-handling routines, but had a measured performance of 40% slower than the hand-written implementation. One reason might be the stalled definition of the constraints beyond the scope of member field declarations, which suggests the semantic evaluation to be performed after filling the entire type with data at runtime. For large types, constraint conflicts regarding anterior fields of the type are supposed to be recognized too late, which makes this strategy inefficient or at least improvable. Despite these drawbacks PacketTypes made the first important steps towards the definition of the problem of parsing binary data and provided a good first approach to face it.

3.2 DATASCRIPT

This work was presented by Godmar Back from the Stanford University at 2002 [Bac02] and is implemented in Java. DataScript is comparable to PacketTypes in many aspects. It adopts parts of the syntax and the two-phased modeling strategy. However, DataScript refines the definition of semantic constraints by moving them to the very location they address, i.e. defines them as member field annotations. In addition, reuse of constraints is enabled with the definition of a parameterized predicate functions, callable from within semantic constraints of the enclosing scope. Finally, Godmar introduces a `forall` operator to allow constraining of array elements (e.g. forcing of numerical order).

Further improvements are made on structural patterns. The tagging- and length-indication patterns are treated in a more intuitive way, by means of constant-assignment and dynamic array allocation as can be seen in listing 3.2. Godmar also introduces support for the location-indication pattern called *labels*, which allows parsing of non-sequential binary files[6]. Moreover, he adds *parameterization* to types, to make them more flexible with regard to certain run-time requirements. The publication defines for instance a *Padding*-type, which dynamically changes its size to pad trailing space of memory blocks with respect to their size and memory alignment.

```
const uint16 ACC_PUBLIC    = 0x0001;
const uint16 ACC_ABSTRACT  = 0x0001; // ...

ClassFile {
   uint32   magic = 0xCAFEBABE;
   uint16   minor_version = 3;
   uint16   major_version = 45;
   uint16   cp_count;
   ConstantPoolInfo  constant_pool[1..cp_count];

   bitmask uint16 ClassFlags {
      ACC_PUBLIC, ACC_FINAL, ACC_ABSTRACT, ACC_INTERFACE, ACC_SUPER
   } acc_flags;

   uint16   this_class  :clazz(this_class);
   uint16   super_class :super_class == 0 || clazz(super_class);
   uint16   interfaces_count;
   {
      uint16   ifidx :  clazz(ifidx);
   } interfaces[interfaces_count];

   uint16      fields_count;
   FieldInfo   fields[field_count];
   uint16      methods_count;
   MethodInfo  methods[methods_count];
   uint16      attributes_count;
   AttributeInfo  attributes[attributes_count];

   constraint  clazz(uint16 x) {constant_pool[x] is cp_class;}
};
```

Listing 3.2: *Excerpt of a Java class specification in DataScript [Bac02].*

The proposed big advantage of DataScript is the ability of its compiler, to generate not only parsing code, but synthesizing code as well, e.g. DataScript-compiled Java code allows reading and writing Java binary class files. However, more important is that DataScript adds error-handling to its parsing routines, involving a back-tracking strategy. Additionally, a weakened kind of the alternatives definition with non-disjoint value ranges introduces the prioritized choice as means to handle unexpected values in a fall-back option.
Finally, the language defines an intuitive and simple syntax for base-2 numbers $\{[01]^{+}\mathrm{b}\}$ making DataScript powerful and convenient enough to express complex structure specifications.

[6].. such as ELF object files. ELF is a standard binary file format for Unix and Unix-like systems.

3.3 Parsing Expression Grammar (PEG)

Parsing Expression Grammar (PEG) is referred to as a *recognition-based syntactic foundation* by its inventor Bryan Ford, who introduced PEG in [For04] at Massachusetts Institute of Technology in the year 2004. PEG emphasizes its difference to the generative system of context-free grammars, by not introducing ambiguity - the reason for complexity of parsing computer languages. Omitting the distinction between scanning and parsing PEG address the formulation of lexical and syntactical structures within the same formal specification language. Its aim to describe *consumption* of input rather than its derivation, in combination with the *prioritized choice*[7] makes PEG-based parsing well suited to machine-oriented languages and protocols. Ford defines *syntactic predicates* `&e` *and* `!e` specifying expected and unwanted input matches with parsing expressions e and subsequent unconditional backtracking, which forms unbounded lookahead altogether.
The set of PEG-expressible languages is described to consist of all deterministic $LR(k)$ languages, including some non-context-free languages. However, unlike CFG depending on the implementation (*recursive descent* or *packrat*), the complexity of either time or space is known to reach exponential order in the worst case [For04].
To a certain extend PEG seems be a promising approach to parsing binary data. In particular, it shares two or more aspects, such as the ordered choice and the unified analysis steps with the approach presented in this thesis. However, it lacks support for sub-symbol encoding, founding again on character granularity as we outlined in clause 2.4. Furthermore, its CFG-alike syntax contradicts our functionality- and usability-requirements being incapable to express the length-indication pattern and having an inconvenient number notation.

3.4 Processing Ad hoc Data Sources (PADS)

PADS [FG05] has been presented by Kathleen Fisher and Robert Gruber (Google) as a project at AT&T Labs Research in 2005. Their motivating aim was to provide a toolset for handling large amounts of *ad hoc data*, such as web server logs, Internet traffic captures or wire formats for telecommunication billing systems. Thus, not only binary data, but also textual and mixed data formats are supported by PADS. In particular a variable *ambient* coding is used to instructs the system to interpret the base numeric types in ASCII, EBCDIC or other customized binary formats.

From its data format specifications a PADS-compiler generates automatically a tool-library for parsing, manipulating, summarizing of the input data and its export to standard formats, such as Extensible Markup Language (XML). Being implemented for application in C, the compiler produces .c- and .h files for compilation by a standard C compiler. Although oriented on the C style syntax and code generation, PADS is claimed not to be restricted to C only, but is rather envisioned to be advanced by other target languages.

[7] in contrast to the nondeterministic alternative in CFGs

As well as the predecessors PacketTypes and DataScript, the PADS language defines numeric base types and structs, unions and arrays for construction of composite types. In order to handle textual input syntactical elements for character and string handling, including support for regular expressions are added. Offering additional handy syntactical shortcuts, such as implicitly typed constants, the PADS language syntax seems to be strongly oriented on DataScript. Also, the use of parameterized types to reduce type diversity and to handle dynamic type representation is adopted. We show a copy of PADS code from the original publication in listing 3.3.

```
Punion client_t {
   Pip       ip;      /- 135.207.23.32
   Phostname host;    /- www.research.att.com
};

Punion auth_id_t {
   Pchar unauthorized : unauthorized == '-';
   Pstring(:' ':) id;
};

Pstruct version_t {
   "HTTP/";
   Puint8 major; '.';
   Puint8 minor;
};

Penum method_t {
   GET, PUT, POST, HEAD, DELETE, LINK, UNLINK
};

bool chkVersion(version_t v, method_t m) {
   if ((v.major == 1) && (v.minor == 1)) return true;
   if ((m == LINK) || (m == UNLINK)) return false;
   return true;
};

Pstruct request_t {
   '\"'; method_t       meth;
   ' ';  Pstring(:' ':) req_uri;
   ' ';  version_t      version : chkVersion(version, meth);
   '\"';
};

Ptypedef Puint16_FW(:3:) response_t : response_t x => {100 <= x && x < 600};

Precord Pstruct entry_t {
         client_t       client;
   ' ';  auth_id_t      remoteID;
   ' ';  auth_id_t      auth;
   " ["; Pdate(:']':)   date;
   "] "; request_t      request;
   ' ';  response_t     response;
   ' ';  Puint32        length;
};

Psource Parray clt_t {
   entry_t [];
}
```

Listing 3.3: *PADS description for web server log data [FG05].*

PADS was designed with robustness in mind, in that the generated code catches and processes system errors, syntax- and semantic errors in a systematic way. During parsing, a canonical representation of the data is instantiated along with a *parse descriptor* containing recorded errors and their characteristics with a reference to the instantiated data. Combined with a nonblocking error-recovery mechanism this allows for application-specific error-management and statistical evaluation.

Another goal of PADS was good performance, which is addressed in a couple of ways. The generated code implements a recursive descent parser (see clause 2.3.4), with multiple entry points, such that parsing can be performed at the different granularity levels. Additionally, the evaluation of semantic constraints is made optional, to help saving run-time costs whenever the evaluation is not required.

All operations realized by the generated code are supported by a shared runtime library. This library offers file I/O and memory management along with other utility functions. Despite that the PADS compiler generates an extensive code base from a data format specification. It is reported that given 68 lines of PADS code, the compiler produces 1432 lines .h file code and 6471 lines .c file code. The expansion is justified by extensive error checking and utility functions. The computation time required compared to a manually implemented reference program is reported to have been halved. Statements about memory consumption are not made.

Based on the work with PADS in the year 2006 Fisher et al. proposed a formal calculus to argue about properties of data description languages in general [FMW06]. This helped in particular to improve and debug PADS.

3.5 binpac

Being part of `Bro`[8] - a unix-based Network Intrusion Detection System (NIDS) binpac represents a collaboration between Ruoming Pang (Google), Vern Paxson and Robin Sommer (International Computer Science Institute) and Larry Peterson (Princeton University). The work was presented in the year 2006 [PPSP06].

The motivation behind binpac was a simplified construction of robust and efficient semantic analyzers to serve the NIDS framework in real-time recognition of malicious packets. Overcoming the lack of abstraction in manual protocol parser implementation, the binpac language and its compiler are related to *yacc* as being a parser construction tool.

Binpac defines a specialized language targeting the work with multi-session application layer network protocols. It is designed to perform *incremental parsing* of multiple concurrent data streams at the same time, distinguishing incoming and outgoing traffic directions. Binpac addresses structural- and indication patterns discussed in clause 2.4.1. Being similarly expressible as PADS, the binpac language offers a system of parameterized types for specification of binary data structures. However, it distinguishes between *explicit* and *implicit type parameters*. The information about the

[8] http://bro-ids.org/

current individual connections and flows is propagated through the type hierarchy as an implicit type parameter without explicit statement. Another modifiable implicit parameter is byte-order (or *endianess*), which allows adjustment to dynamic byte-order changes at run-time, demanded by some communication standards[9]. This enables byte-order independent specification of data structures.

```
...
extern type BroConn;
extern type HTTP_HeaderInfo;
%header{
   // Between %.*{ and %} is embedded C++ header/code
   class HTTP_HeaderInfo {
      public:
         HTTP_HeaderInfo(HTTP_Headers *headers) {
            delivery_mode = UNKNOWN_DELIVERY_MODE;
            for (int i = 0; i < headers->length(); ++i) {
               HTTP_Header *h = (*headers)[i];
               if ( h->name() == "CONTENT-LENGTH" ) {
                  delivery_mode = CONTENT_LENGTH;
                  content_length = to_int(h->value());
               } else if ( h->name() == "TRANSFER-ENCODING"
                  && has_prefix(h->value(), "CHUNKED") ) {
                  delivery_mode = CHUNKED;
               }
            }
         }
         DeliveryMode delivery_mode;
         int content_length;
   };
%}

connection HTTP_Conn(bro_conn: BroConn) {
   upflow = HTTP_Flow(true);
   downflow = HTTP_Flow(false);
};

flow HTTP_Flow(is_orig: bool) {
   flowunit = HTTP_PDU(is_orig) withcontext(connection, this);
};

type HTTP_PDU(is_orig: bool) = case is_orig of {
   true     -> request:   HTTP_Request;
   false    -> reply:     HTTP_Reply;
};
type HTTP_Request = record {
   request:    HTTP_RequestLine;
   msg:        HTTP_Message;
};
type HTTP_Reply = record {
   reply:   HTTP_ReplyLine;
   msg:     HTTP_Message;
};
...
```

Listing 3.4: *Excerpt of a HTTP parser in binpac [PPSP06].*

Finally, there exists an interesting and powerful feature in binpac. In analogy to *yacc* binpac allows embedding of C++ code blocks, as an interface to interact with the host language. By declaration of *external* types a binding to existing implementations

[9] such as DCE/RPC: "Distributed Computing Environment / Remote Procedure Calls"

can be established, merging the powerful expressiveness of a general purpose language with the compactness and simplicity of the declarative data format grammar. This facility puts binpac clearly beyond the other works, enabling encapsulation of the entire required functionality within one code file with the option to reuse existing parser code (hand-written or generated) if necessary.

Compared with manual reference implementations the generated code sizes are reported to amount just 35-50% lines of code while performing about 17% faster with a comparable memory consumption. Offering a robust performance binpac-generated parsers process the HTTP protocol at a throughput of 298 Mbps.

3.6 GAPA

The final project in our review emerged at Microsoft Research in cooperation with IEEE, the Carnegie Mellon University and UC Berkley in 2007. In [BBW+07] Borisov et al. presented GAPA - a Generic Application-Level Protocol Analyzer and its Language, which culminated in the *Forefront Threat Management Gateway (TMG)*[10] - a Windows-based implementation of a NIDS.
GAPA shares with binpac the same goal of application-level protocol analysis. However, Borisov et al. took a different approach to achieve this. In contrast to binpac, GAPA was intended for usage in end-point firewalls rather than network gateways. The main aspects targeted by GAPA were safety, real-time analysis and rapid development of efficient analyzers. Hence, the only acceptable way of realization for the authors was the use of a memory-safe language for interpretation at run-time rather than compilation. As a result, the language had to provide all the necessary elements required for complete protocol description and specification of analysis procedures, without need of external data processing logic.

In GAPA a protocol specification is segmented into parts with respect to distinct concerns they address. For structure specification in the grammar section it uses BNF-like rewrite rules augmented by an internal C-like script language.
This script language provides enough expressiveness to perform frequently required computations, however it is rather restricted to avoid self-caused run-time errors and efficiency issues. It is strictly typed, performs bounds checks on array access and has no dynamic memory allocation. Furthermore, the only loop constructs it provides is a for-each-loop which can not be nested.

The analysis process is controlled by a state-machine simulation specified in the corresponding segment. Tuples of states and possible traffic directions designate scripted *handlers*. After parsing the corresponding data by means of a recursive descent parser the associated handler is triggered to determine a succeeding state. This explicit state machine modeling gives GAPA the expressive power to realize rather complex interaction scenarios.
Although claimed to support binary data structures, the GAPA language seems to be

[10] http://www.microsoft.com/forefront/threat-management-gateway/en/us/default.aspx

designed in favor of textual protocol specification. Borisov et al. incorporate syntax directed parsing we described in clause 2.3.5 to implement handling of indication patterns for the purpose of parsing binary data. Additionally, support for k-bit integers, dynamic array notation and structural patterns realized in BNF are offered.
We show the HTTP specification written in GAPA in listing 3.5.

```
protocol HTTPProtocol {
  transport = (80/TCP);

  /* Session variables */
  int32 content_length = 0;
  bool chunked = false;
  bool keep_alive = false;

  /* message format specification
  in BNF-like format */
  grammar {
    WS = "[ \t]+";
    CRLF = "\r\n";
    %%
    HTTP_message -> Request | Response;
    Request -> RequestLine HeadersBody;
    Response -> ResponseLine HeadersBody;
    HeadersBody ->
      {
        chunked = false;
        keep_alive = false;
        content_length = 0;
      }
      Headers CRLF
        {
          /*message_body's type is resolved
          (:=) at runtime based on
          Transfer-Encoding */
          if (chunked)
            message_body := ChunkedBody;
          else
            message_body := NormalBody;
        }
      message_body:?;

    Headers -> GeneralHeader Headers|;
    GeneralHeader->
      name:"[A-Za-z0-9-]+" ":"
      value:"[^\r\n]*" CRLF
      {
        if (name == "Content-Length"){
          content_length=strtol(value,10);
        }else if (name == "Transfer-Encoding"
          && value == "chunked") {
            /* slight simplification */
            chunked = true;
    ...
    ...
        } else if (name == "Connection"
            && value == "keep-alive") {
              keep_alive = true;
        }
      };
    NormalBody ->
      bodypart: byte[content_length]
      {
        /* "send": sending "bodypart" to
        the upper layer (e.g., RPC)
        for further parsing */
        send(bodypart);
      };
    [...]
  };// Grammar

  state-machine httpMachine
  {
    (S_Request,IN) -> H_Request;
    (S_Response,OUT) -> H_Response;
    initial_state = S_Request;
    final_state = S_Final;
  };

  /* Always expect a response after a request */
  handler H_Request (HTTP_message){
    int headerCount = 0;

    /* visitor syntax */
    @GeneralHeader->{
      print("header name = %v\n", name);
      headerCount++;
    }
    print(''Total number of headers: %v\n'',
      headerCount);

    return S_Response;
  };

  handler H_Response(HTTP_message){
    if (keep_alive){
      return S_Request;
    } else {
      return S_Final;
    }
  };
};  // protocol
```

Listing 3.5: *HTTP specification in GAPA language [BBW+07].*

3.7 Summary

Covering the past decade the construction of binary parsers from domain-specific languages has been explored in a couple of research projects in collaboration between scientists and the IT-industry. The results discussed here share the idea of using types known from general purpose programming languages for description of data structures with extensions allowing definition of semantic constraints. The approach to use a domain specific language to improve the development of binary parsers has led to positive results not only in terms of time and efficiency, but also in terms of quality. We conclude this section with a short listing of encountered approaches related to formal specifications and communication protocols.

Protocol Design

LOTOS: Specification language for protocol design and verification.

SDL: Specification and Description Language for communication modeling.

ASN.1: Abstract Syntax Notation One for high-level structure representation.

Promela++: Formal language for construction of correct and efficient protocols.

Implementation

StateCharts: FSA-based programming of communication protocols and systems.

Esterel: FSA-based programming of communication protocols and systems.

Prolac: Language for modular implementation of networking protocols.

APG: An ABNF parser generator.

RTAG: Real-Time Asynchronous Grammars for specifying protocols.

PADS/ML: A functional data description language.

Parsing and Analysis

PacketTypes: Type system for networking packet specification.

PEG: Parsing Expression Grammar.

binpac: Application protocol parser generator (part of Bro).

GAPA: Application protocol parser generator (part of Microsoft Forefront TMG).

Zebu: Application protocol parser generator.

Parsing and Manipulation

DataScript: Type system for parsing and manipulation of binary data.

PADS: Framework for construction of ad hoc data management tools.

DFDL: XML-based Data Format Description Language.

Protocol Recognition

Discoverer: Automatic Protocol Reverse Engineering tool (Microsoft)

PADS: Framework for construction of ad hoc data management tools.

4 Design Concept

Starting with a thorough analysis of requirements to be stated on the language, its compiler and the generated binary parsers, we outline important aspects to be considered in the project realization. In the subsequent clauses 4.2 through 4.4 we present the solution-design which aims to cover the stated requirements enclosing the design of the language, the compiler architecture and the model of binary parsers to be generated by the compiler.

4.1 Requirement Analysis

In this section we are going to inspect and outline the requirements of a system for construction of binary parsers as to define quality directives for its implementation. Some of the requirements were mentioned directly or indirectly during the previous clauses. Nevertheless, they will be stated here explicitly once more along with other requirements unmentioned earlier. We remind the reader of the conclusion made with regard to the transcription problem in clause 2.2.3 on page 13, to support the developer in bridging the declarative and operational representations of specification code. With the insights from the previous clauses the concept to achieve this can be sketched. The new approach in figure 4.1 involves a declarative *domain specific language* for the specification of data structures. These serve then the compiler as blueprints for the automated construction of binary parsers.

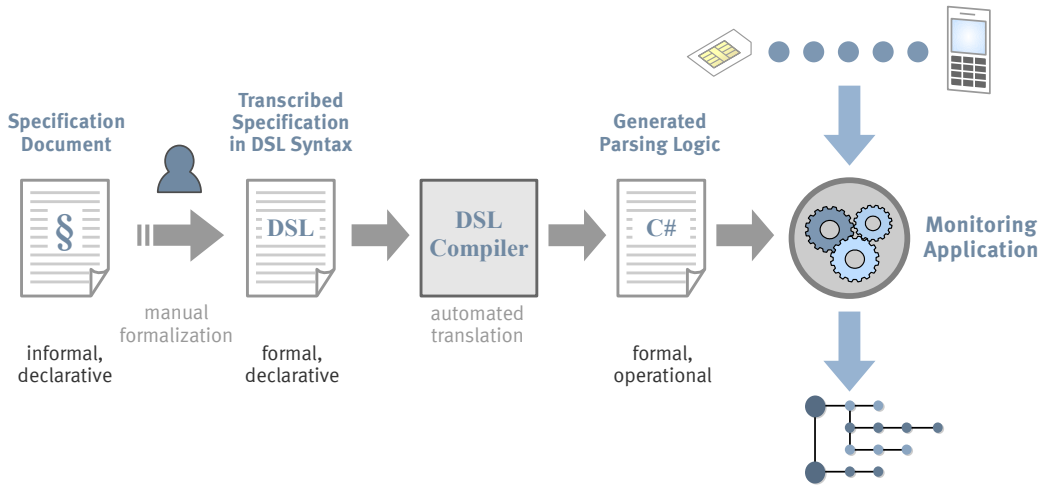


Figure 4.1: *By means of the declarative specification language the developer can easily transcribe specifications to formal declarative code. From this formalized specification the compiler then generates C# code of a binary parser for the specified data structures. C# is an external requirement specified in clause 4.1.4.*

Within this concept three parts, the specification language, the compiler and the generated binary parsers contribute to the overall quality and the successful application of the concept. Each of these integral parts has its characteristic objectives whose

quality requirements must be identified. We formulate the requirements with regard to the quality characteristics of ISO 9126-1 through 9126-4 quality model, which are subdivided into *internal*, *external* and *in-use* quality characteristics [ISO01, ISO03a, ISO03b, ISO04].

4.1.1 Language Quality

The requirements on the aspired specification language are primarily based on *external*- and *in-use* quality characteristics. Since the language is nothing more than a form of information representation, the internal quality is achieved solely by its compiler. We identified the characteristics of *usability*, *maintainability* and *functionality* to contribute most to the language quality.

Functionality *(Expressiveness, Accuracy, Suitability)*

To enable concise transcription the language must have the capability to express all necessary forms and sizes of structural patterns, values and value ranges used in specification documents. It must provide syntactical means to augment the defined structure with semantic constraints as to enable modeling of indication patterns and other custom semantic relations. Furthermore, a convenient notation of binary numbers[11], must be provided as naturally as is done with octal, decimal and hexadecimal numbers in common languages. Hence, complex symbol escaping patterns must be avoided.

Usability *(Understandability, Convenience)*

To help programmers new to the transcription task to use the language, commonly known syntactical elements from other programming languages shall be reused as far as possible. An easy to read but yet compact notation as presented in figure 2.19, resembles the tabular representation from standard documents we showed in the definition of an APDU in table 2.1. Thus, it is assumed to leverage transcription, making it a rewrite of tables in C like syntax.
The language shall foster reuse of definitions for extension and modification through modularity, as to minimize redundancy and code size.
To avoid an artificially imposed spatial order on named data structures, which possibly contradicts the order of definitions in the standard document, the implementation of the Algol scope rule is expected to be a clear advantage[12].

Maintainability *(Analyzability, Changeability)*

In [FG05] Fisher et al. understand the specification code as a *living* documentation, which not only serves as input for the compiler, but also as a readable, formal document which is subject to changes. This suggests inclusion of syntactic elements for meta-information embedding, such as references to standard documents, version numbers

[11] Protocol specifications make heavy use of it.
[12] We explained the rule in clause 2.3.5

and other arbitrary data, to allow identification and comparison of compiled binary parsers. Furthermore, the attachment of meta-information to specified data structures could be used to pretty-print parsed data in terms of the normative document, which also aids in localization of faults in the code (*debugging*).

4.1.2 Compiler Quality

The quality characteristics of the compiler regard nearly all aspects of software quality. The universal characteristics of any compiler include *correctness*, *efficiency*, *code efficiency*, *user support* and *robustness* as stated in [Kas09b]. In our case, the compiler plays the role of a parser generator, which synthesizes high-level language code. This relieves its implementation ceding low-level code optimization and linking to the C# compiler.

Functionality *(Correctness)*

Correctness of compilation is the consistent translation of a program from the source language to the target language without modifications of the specified meaning. For correctly specified source code the compiler must produce target code which exhibits exactly the very properties and behavior specified in the source code. No correct specification may cause errors, and every incorrect specification must produce at least one error during compilation [Kas90].

Reliability (*Robustness*, *User Support*)

In case of incorrect input, detailed informative messages must be reported to the user, helping to locate and to identify the exact position and possible causes of the fault in terms of the source code file. Ideally the compiler does not crash under any circumstances.

Efficiency (*Time and Memory Consumption*)

Using established construction tools for the development of the compiler, its implementation is expected to be efficient enough in terms of time and memory consumption for execution on every common Windows-based computer system today. No extra resources shall be required.

Maintainability

Compiler construction is a well-investigated and well-understood area, for which appropriate architectural patterns have evolved over time. A consistent separation of compiler modules for distinct concerns leads to a clear understanding of functionality, enhancing the maintainability and extendability. Standardized methods and a reasonable compiler architecture are essential for this work to achieve the goal and provide a compiler of good quality.

Portability (*Retargeting*)

Since the front-end of the compiler is to be generated from the language specification, the choice of the construction tool and the form of specification has a great impact on the portability of the constructed compiler. At the moment, the only demanded platform to run at is Microsoft Windows XP or higher, which must be supported by the generators.

4.1.3 Quality of Generated Parsers

The binary parsers generated by the compiler are thought to be running as integral parts of monitoring applications. Hence, major in-use quality characteristics are associated to the overall performance of the generated parsers.

Functionality (*Process-oriented Parsing*)

Parsing data for the purpose of monitoring raises the importance of the process quality above the quality of a certain parsed data packet. That is, for a large set of communication packets and the steadily growing number of data structures, support for parsing the entire set completely is an ideal aim, which, however, can hardly be reached. This is a weakened requirement compared to the demanded total correctness of a typical compiler front-end. The continuity of the parsing process itself is much more important. It has to be aligned towards constant data streaming, including appropriate configuration of the input- and output buffers (*pipes and filters architecture*). The system must be capable to parse data excerpts partially, suspending upon absence of input without blocking the entire application, and resuming on arrival of new input data.

Reliability (*Robustness*)

A related property required in the implementation is fault tolerance and robustness. Not only the resistance to malformed, lost or unknown input data, but the reasonable statement on the exact error location and its probable reasons is essential for debugging the testees of the monitoring system. The application must safely recover to a normal operation, whenever inconsistencies occur.

Efficiency (*Online Performance*)

Finally, the parsing is expected to be performed online, hence a good performance level is aspired. Especially in case of invalid data the recovery time is assumed to achieve critical values. However, a normal operation can be performed in real-time as known from the current manual implementation.

4.1.4 External Requirements

Implementation Language

One external requirement from COMPRION is that the implementation of binary parsers has to be based on C#. The reason is the transcendental use of the Microsoft .NET Platform and C# in all new products.

License

Another requirement concerns the potential commercial use of the outcome of this thesis. It prohibits employment of tools or source code published under license agreements which enforce publication of the product internals.

Now that the important requirements of a good solution have been identified, we are going to focus on the actual design developed with regard to these requirements.

4.2 The PBPG Language

The Paderborn Binary Parser Generator language syntax of is mainly oriented on DATASCRIPT, allowing convenient definition of types and their semantic constraints. The pivotal syntactic structure is the type definition, which has the form `NewType BaseType`, where `NewType` is a defining occurrence and `BaseType` an applied occurrence of the type names. The name resolution follows the Algol scope rules. We discuss the major aspects of the language syntax in the following paragraphs. The entire specification can be found in the Annex A.2.

Primitive Types The built-in primitive data types provide the atomic units for the construction of composite types. In order to address the requirement of bit-level structure-modeling, we define not only the common numeric types BYTE, SHORT, INT, LONG, but also the sub-byte types BITx1, BITx2, ... , BITx7. Together with the composition elements these should suffice to describe data of arbitrary granularity. The convenient notation of numbers consists of decimal, binary, octal and hexadecimal forms, where the latter three use the short prefixes `#`, `0` and `$`. For example, the decimal number `12` could also be written as `#1100`, `014` or `$C`.

Composite Types In analogy to the discussed languages of the previous section the construction of composite types is achieved by definition of structs, unions and arrays. Aiming for simplicity and compactness of notation we chose braces `{..}` for structs, angle brackets `<..>` for unions and brackets `[..]` for arrays. They are used much like in C, however the leading keywords *struct* and *union* are omitted. In the figure 4.2 we show the declaration of a composite type in terms of PBPG.

```
U1 <
   Alt1 {
      tag   $0F;
      data  INT;
   };
   Alt2 {
      tag   $FF;
      data  BYTE;
   };
   Alt3 {
      tag   BYTE;
      data  BYTE[4];
   };
>;
```

Figure 4.2: *PBPG composite type specification incorporating struct, union and array declaration.*

Whereas the notion of structs and unions is straight forward, the concept of arrays as designed for PBPG requires explanation. An array in PBPG is defined as a repetition of the array-element type, restricted by range boundaries or a semantic constraint.

This realizes not only the repetition operations from table 2.12, but also introduces the notion of a *conditional* array as a consequent application of the binary parser model we concluded in figure 2.18. In particular array definitions in PBPG can take on the following forms, where `n,m` represent integer values and the constraint `c` a Boolean expression as explained in the following clause.

`A1 ElementType[n..m]`	bounded
`A2 ElementType[n..]`	left-bounded
`A3 ElementType[m]`	constant
`A4 ElementType[]`	optional
`A5 ElementType[:c]`	conditional

At run-time these arrays consume input as long as the boundary conditions are not violated or the stated semantic condition holds respectively. In case of `A1`, this means if less than `n` elements have been successfully parsed, i.e. one element did not match the expected element type, the entire array has to be regarded as *not matching*. On the other hand after `m` successful iterations parsing of the subsequent type within the type structure is automatically started.

The definition of arrays requires careful handling and the knowledge of the consequences during parsing. It is obvious that for some structure specifications ambiguity can be created. For example:

```
ambiguous {
   list  BYTE[0..3];
   term  255;
};
```

For the input `255 255 255` there exist multiple correct assignments of input bytes to the fields `list` and `term`. At this point we refer to the argumentation in clause 2.4 where we pointed out that the specified data structures are to be processed by comparably simple devices and that the parsing process has to be *guided* by the semantic constraints as to be deterministic. This means that no correct (or correctly interpreted) specification document does employ such data structures because of the ambiguity.
Nevertheless, in this very example we can show, that by using an additional type we can disambiguate the structure specification:

```
unambiguous {
   list  NonTerm[0..3];
   term  255;
};

NonTerm BYTE: NonTerm != 255;
```

At the moment, the user has to pay attention to such situations. They might indicate an inconsistency with the standard document or even a problem within the document itself. However, in the future automatic recognition of ambiguity might become part of the compiler implementation.

Constraints The definition of a type may be padded with a Boolean expression headed by a colon - the semantic constraint (or post-parsing condition) of the type. Right after reading in the data for the defined type this condition evaluates the classification success. Operands of the expression can be the members of types defined prior to the textual location of the constraint, inside the type definition scope. In order to allow more sophisticated computations, externally defined code can be accessed through a function call as well. The signature of the called functions has to be compatible with the operators of the semantic constraint expression, as to be translatable to a correct host-language representation. In the next example, the called function `Checksum(data)` must have the expected LONG return type, whereas the next function `PostValidation(data, len)` must have a Boolean return type. Furthermore, the definition of both functions must be accessible in the scope of their applied occurrences in the translated representation.

```
DataChunk {
   len   BYTE: 0 < len;
   data  BYTE[len];
   cs    LONG: cs == Checksum(data);
}: PostValidation(data, len);
```

Using constraints in conditional arrays, constructs such as null-terminated strings or padding fields can be modeled. The meaning of such a semantic constraint is that of a parsing condition[13]. As long as it holds, the array is continued to be parsed, otherwise the process stops, yielding an array of those elements only that fulfill the parse-condition. Here the symbol `@` serves as a reference to the currently parsed value to be assigned to the array element.

```
ntChars byte[:@!=0]          null-terminated string
padding byte[:@==$FF]        padding
```

[13] In contrast to the post-parsing condition the parsing condition is checked not only once but for each of the array elements.

4.3 The PBPG Compiler

For the proposed specification language the PBPG has been developed, whose name was motivated by the generating tools used for its construction with regard to the central aim to generate binary parsers. On the next pages we depict the semi-generated architecture of PBPG. From the construction tools on we focus on design aspects and the stages of compilation from analysis to synthesis.

4.3.1 Choice of Tools

For the construction of the compiler front-end several construction tools came into question. Among a great number of generators such as ANTLR, yacc, Bison, COCO/R and integrated compiler construction systems including ELI[14], GENTLE and COCKTAIL[15] the choice fell on the generators GPLEX and GPPG which turned out be very suited to the set of requirements of this project.

Gardens Point Scanner Generator (GPLEX) and Gardens Point Parser Generator (GPPG) originated at Queensland University of Technology as contributions to the generators MPLEX and MPPG of the Visual Studio Managed Babel distribution by Microsoft. Developed by Wayne Kelly and John Gough GPLEX and GPPG form a collaborative pair of tools for construction of LALR(1) parsers and their scanners. The tools are .NET based versions of *LEX* and *yacc* which gained particular importance on many Unix systems as standard tools for compiler construction. Being developed entirely in C# GPLEX and GPPG are total re-implementations of their conceptual predecessors and are released in open source form under "Free-BSD" style license arrangements. Both generators produce C#-based automata implementations designed to work together. However, also a successful combination of generated code with handwritten scanners and parsers and those generated by COCO/R is reported[16].
We summarize the advantages of GPLEX and GPPG in the following list:

- **State of the art technique:** Support for LALR(1) parsers.
- **Reliability:** The generators are certain to be tested thoroughly by Microsoft.
- **Single development environment** Implemented in the same language (C#) as the binary parsers are demanded to be.
- **Code reuse**: License arrangements allow free reuse of source code for commercial purposes.
- **Ease of use:** Generators can easily be integrated into the build process of Visual Studio.

[14] An integrated Toolset for Compiler Construction. Eli is being developed and maintained by researchers from the University of Paderborn, University of Colorado (USA) and the Macquarie University (Australia). Eli is freely available at http://eli-project.sourceforge.net

[15] GENTLE and COCKTAIL emerged in the late eighties at the GMD Lab at University of Karlsruhe and are now independent compiler construction systems. Please refer to http://gentle.compilertools.net/ and http://www.cocolab.com/

[16] Freely available at http://gplex.codeplex.com/ and http://gppg.codeplex.com/

However, there exist drawbacks in the choice of the generators too.

- **No Attribute Grammar:** There is no support of attribution in grammars provided by systems like ELI.
- **Analysis-only solution:** Implementation of static semantics analysis and code synthesis are totally ceded to the developer.

These drawbacks have to be overcome during implementation, meaning that more effort has be to spent on the manual realization.

4.3.2 Front-end

The analysis front-end of the PBPG compiler can be described best as an integration of parts from three distinct sources. First, the automata for the lexical and syntactical analysis are generated from grammar specifications. Hence, their internal architecture and the interconnection are dictated by the generators. Second, the generic parts like the error-handling mechanism and the input-buffer implementation have been adopted from the GPPG generator source code. The third part finalizing the architecture is most project specific and developed from scratch. It manages the construction of the abstract syntax tree and its analysis with regard to static semantics. We show the logical view of the entire front-end in figure 4.3.

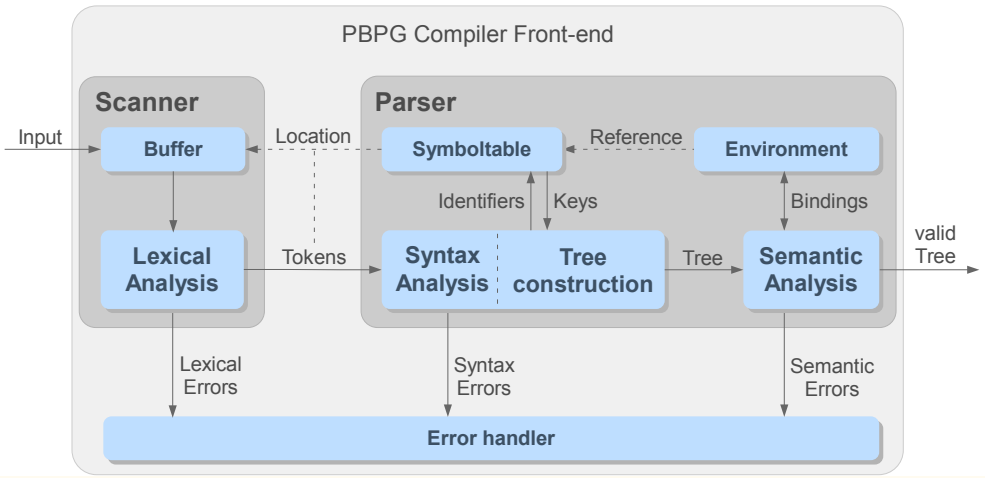


Figure 4.3: *Logical view of the analysis front-end of PBPG. The parser employs a Symboltable and an Environment module for name handling, whereas all compiler parts attach to a common Error handler for emission of errors, warnings and status information.*

This model shows three noteworthy design aspects. First, all stages of compilation including the back-end not shown here refer to the same error handler which is the central module to collect, store and log errors, warnings and other status information throughout the entire application.
Second, each token from the scanner is associated with a location reference pointing to the position and length of the token's character sequence in terms of the input buffer. Thus, any output message regarding the input document can be expressed based on the original wording and exact textual position.
Third, during syntax analysis the encountered identifiers are stored in a symbol table along with their defining- or applied occurrence indication, which yields a unique numeric key representing the identifier. In the subsequent name analysis these keys are used for the construction of scopes, name binding and the verification of correct applied occurrences.
Starting with the architecture of the generated core elements and the interface between the scanner and the parser we are going to focus on details of this model.

The Generated Core

In order to establish a functional, yet loose coupling between the scanner and the parser, a sort of *decorator design pattern* realizes the scanner-parser interface as shown in figure 4.4.

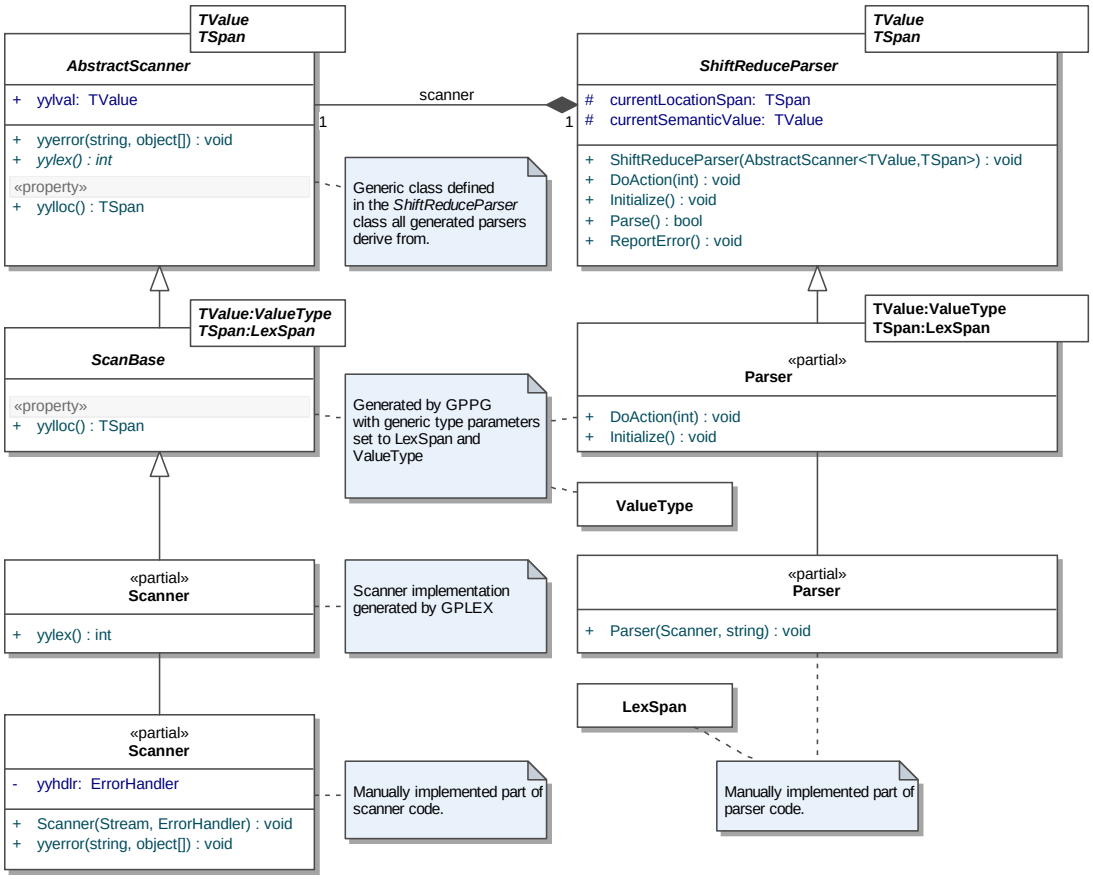


Figure 4.4: *The scanner-parser interface is realized using the decorator design pattern. GPPG generates the parser automaton and the abstract ScanBase class to derive from in the scanner implementation for this parser.*

The definition of the class *AbstractScanner* and the implementation of the invariant parser automaton *ShiftReduceParser* is provided by the GPPG implementation. Both are defined as templates (or *generics* in C# terminology). From the grammar file GPPG creates the derived partial[17] *Parser* class which contains the state table and the semantic actions for the *ShiftReduceParser* to operate on. Moreover, it generates the abstract *ScanBase* class - the skeleton for deriving scanners and a *ValueType* structure, which

[17] C# allows the implementations of classes, interfaces and structs to be split into multiple parts which collectively define the one entity. The separation in a generated and a manual part is very practical as it enables use of the editing support like *syntax highlight* and *error hinting* of the C# code editor. In class diagrams we represent the *partial* modifier as a stereotype.

serves as a broker for values passed from the scanner to the parser. It is assigned to *TValue* in the deriving classes. The location reference type *LexSpan* is not generated, but must be provided by the actually used input buffer implementation. However, since this part of code as well as the *ErrorHandler* class are already available in the GPPG's own codebase it was integrated into the code of PBPG. Finally, the GPLEX-generated scanners as *ScanBase* descendands are endowed with an additional manually defined constructor in the other (partial) *Scanner* class which is required for assignment of an *ErrorHandler* instance and the input stream object.

The Scanner Specification

The GPLEX-generated scanner implements a table-driven FSA. Like LEX-based scanners this class offers the function *yylex()* to be called by the parser on demand of the next token, and *yyerror(..)* for the emission of errors to the common error handler. The members *yylloc()* and *yylval* are of the types *LexSpan* and *ValueType*. For each recognized token *yylloc()* designates the corresponding text span in the input, whereas *yylval* designates the token's semantic value. In the following figure an excerpt of the scanner specification is shown, which clarifies the usage of *yylval* and *yylloc()*.

```
//== DEFINITIONS SECTION ===================================
DecDig   [0-9]
OctDig   [0-7]HexDig [0-9a-fA-F]
...

Bin      #[0,1]+        // # prefix
Oct      0{OctDig}+     // 0 prefix
Dec      {DecDig}+
Hex      ${HexDig}+     // $ prefix
...

%% //== RULES SECTION ========================================
{Bin}    {yylval.iVal = ParseNum(yytext.Substring(1), 2);
         return (int)Tokens.INTEGER;}

{Hex}    {yylval.iVal = ParseNum(yytext.Substring(1), 16);
         return (int)Tokens.INTEGER;}
...

// Scanner epilog
%{
   yylloc = new LexSpan(tokLin, tokCol, tokELin,   tokECol,
                        tokPos, tokEPos, buffer);
%}
%% //== USER CODE ============================================
// located in the partial class
```

Figure 4.5: *Excerpt of the PBPG scanner specification.*

The specification file is divided into three sections. In the "Definitions" section regular expression based character classes and patterns are defined. Using these patterns in the "Rules" section an ordered list of token definitions is stated. The definitions on the left are augmented by *semantic actions* enclosed in braces. These represent C# code blocks which are copied into the generated implementation of the FSA. At runtime the semantic actions are executed whenever the associated token is recognized. Each semantic action returns a number identifying the token. For certain tokens, like integer or string literals it might be necessary to define the token's semantic value. In the definition of the *Bin* and *Hex* tokens this requires a string-to-integer translation, according to the right base. The parser can then access the translated numeric value through *yylval*. At the bottom of the "Rules" section another code block contains the assignment of a LexSpan object to the *yylloc* field. At any time the currently matched text span is computed and written to *yylloc* for the use by the parser. The last section may contain further arbitrary user code to be copied into the scanner implementation. However, we do not make use of it due to availability of partial class definition. Detailed information about the structure of the input document and its mapping to the scanner code can be found in the GPLEX documentation [Gou09a].

The Parser Specification

The GPPG grammar specification exhibits the same *yacc*-style sectioning as the scanner specification. The generator produces a table-driven LALR(1) parser, whose functionality is spread over the class hierarchy as shown in figure 4.4. Its invariant code resides in the *ShiftReduceParser* class, whereas the grammar-specific tables and semantic actions are maintained in the concrete *Parser* class of the GPPG output.

The first section of the specification document enumerates the tokens used in the subsequent rules section. By means of `%union` it defines the *ValueType* structure from our class diagram in figure 4.4 as the *TValue* generic type parameter. This structure specifies variables to hold the semantic values of tokens provided by the scanner, and of the nonterminals evaluated in their semantic actions. The variables are bound to the grammar symbols by means of `%token` and `%type`, as can be seen in the example figure 4.6. The start symbol of the grammar is designated by `%start`.

In analogy to the semantic actions of the scanner specification and the described syntax-directed translation in figure 2.3.5 C#-based code blocks can be associated to every production. Again, copied into the generated parser code these semantic actions are prepared to be triggered during parsing whenever the associated production is applied. Using the symbols `$$` and the numerated versions `$1`, `$2`, ... the assigned semantic values of the left-hand side nonterminal or the others to the right of ':' can be addressed. The symbols `@$` and `@1`, `@2`, ... represent textual locations of the production symbols, i.e. their *LexSpan* instances which are created at the bottom of the scanner specification. In both cases (`$n` and `@n`), the number n refers to n-th symbol to the right of ':', whereas `$$` and `@$` refer to the single left-hand side production symbol.

Due to the nature of bottom-up parsing the semantic values of the right-hand side

symbols of the production are available at the time of reduction. They can contribute to the definition of the production's left-hand side symbol value. As can be seen in the example the final reduction performed creates the root `UnitNode` of the syntax tree constructed through semantic actions during the reduction process.

Same as for the scanner specification the user-code section remains empty, due to a more comfortable definition of partial classes. In figure 4.6 we present a selective excerpt of the PBPG language grammar, showing the most interesting aspects of a GPPG specification. The entire document is located in the annex in clause A.2.2. Further information and detailed descriptions on GPPG specifications can be found in [Gou09b].

```
//== DEFINITIONS SECTION ===================================
%YYLTYPE LexSpan
...

%union {
    public long iVal;
    ...
    public AstNode node;
    public TypeNode typeNode;
}

%token   <iVal> INTEGER "Integer value"
%left    <tkn> BIPIPE "||" // left-associative operator
%right   <tkn> EXCL "!"    // right-associative operator
...
%type    <typeNode>  TypeDecl Scheme
%type    <node>      Unit UnitHdr

%start   Unit

%% //== RULES SECTION =======================================
Unit     : UnitHdr Includes Imports TypeDecls {
             $$ = new UnitNode($1 as IdentNode); // root of AST
             ($$ as UnitNode).AddTypeDecls($4);
             spec.Add($$);}
         ;
...
TypeDecl : IDENT Scheme Repetition Constraint SEMI {
             $ = new TypeNode(@1, $2 as TypeNode, // type
                $3 as RepetitionNode);
             $$.SetIdent(new IdentNode(@1, true));
             $$.SetConstr($4 as ConstraintNode);}
         ;
...
Expr     : OpLogicOR {$$ = $1;}
         ;
OpLogicOR : OpLogicOR BIPIPE OpLogicAND {
             $$ = new ExprNode(@2,$2,$1,$3);}
         | OpLogicAND {$$ = $1;}
         ;
...
%% //== USER CODE ========================================
// located in the partial class
```

Figure 4.6: *Excerpt of the PBPG language grammar specification.*

Syntax Tree Construction

Controlled by the bottom-up parsing process the construction of the syntax tree is a rather simple process. For each production application object instances are created to store and represent the relevant parsed information, subordinating the previously created object instances. The central data structure to implement such tree construction defines a tree node and associations to a parent and child nodes. From this ancestor for each syntactical element of the PBPG language specialized derivatives with adequate internal organization are defined. The following table outlines the most relevant syntactical elements and their C# classes, all of which are descendants of the common `AstNode` class. A complete list of node classes of the PBPG parser can be found in the class diagram in figure 4.8.

Syntactical Element	Content	Dedicated C# class
Specification document	A list of globally defined data structures and normative meta-information	`Specification`
Type definition	The kind of substructure, i.e. struct, union, array or plain	`TypeNode`
Semantic Constraints	A Boolean expression	`ConstraintNode`
Array description	Element-type and element repetition information	`RepetitionNode`
Expressions	Arithmetic and Boolean expressions from semantic constraints and array definitions	`ExprNode`
Type references	Type inheritance information	`TypeRefNode`
Native Types	Occupied Bit sizes	`NativeTypeNode`
Identifiers	Type names for name analysis	`IdentNode`, `QIdentNode`
Constants	Occupied Bit size and value	`IntNode`, `BoolNode`

Given this palette of nodes classes an abstract syntax tree as depicted in Figure 4.7 could be constructed during parsing the following exemplary input.

```
unit MySpec;

aUnion <
   aValue1  BYTE;
   aStruct {
      aRef     arrType;
      aValue2  BYTE: aValue2 > 0;
   };
>;
arrType BYTE[10];
```

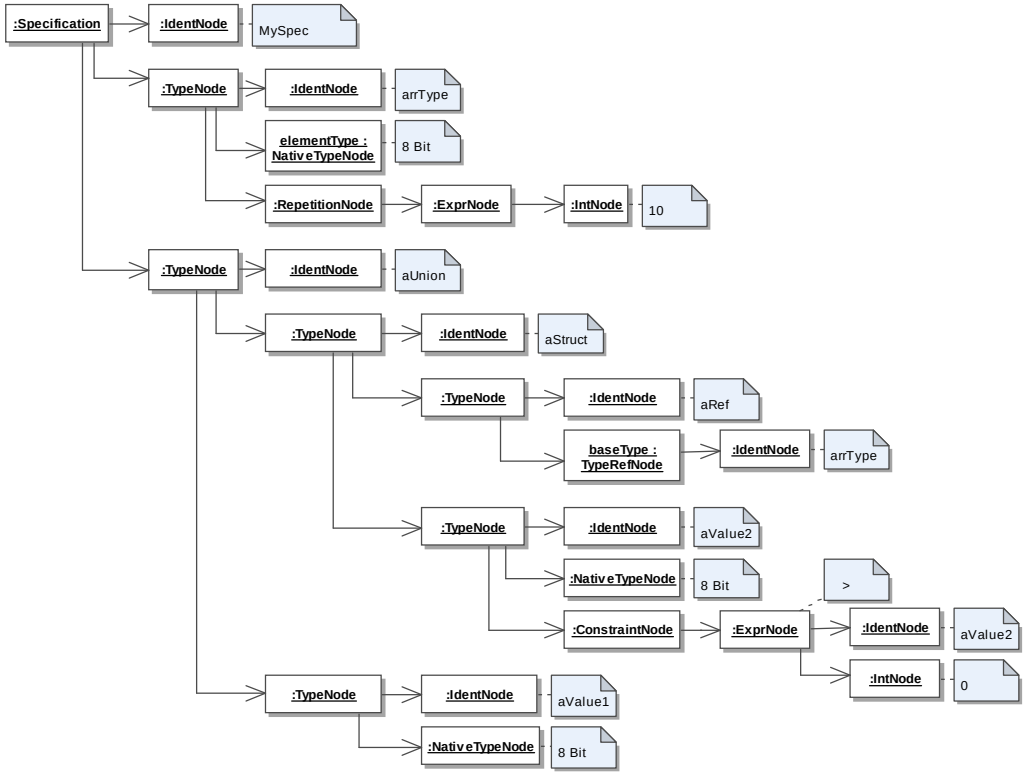


Figure 4.7: *The information of the input document is condensed in the object tree of specialized node-class objects. Such trees can obviously grow quite large.*

An important aspect in this tree construction procedure is that the encountered identifiers and their application (definition or use) may already be stored in the symbol table, such that the subsequent name analysis can operate on identifier keys rather than on the identifiers. By definition of the PBPG language the only possible defining occurrence of an identifier is the left-hand side identifier of type definition. This regards every *IdentNode* instance being child of a *TypeNode*. Every other occurrence can only represent an applied occurrence. Along with the actual text span the identifier is stored in the symbol table without any further analysis for the moment. For the key calculation a generic *Dictionary<string, int>* class is employed, such that for each new input string not part of the dictionary a new unique integer value is assigned and stored in the dictionary. For input strings contained in the dictionary the associated stored number is returned. As a result, two different occurrences of the same textual string yield the same key.

Another *Dictionary<int, IdentInfo>* is maintained to capture and count every instantiated *IdentInfo*, such that each new instance is endowed with its own symbol number for later access.

We present all the data structures involved in the construction of the abstract syntax tree and their correlation in the class diagram figure 4.8.

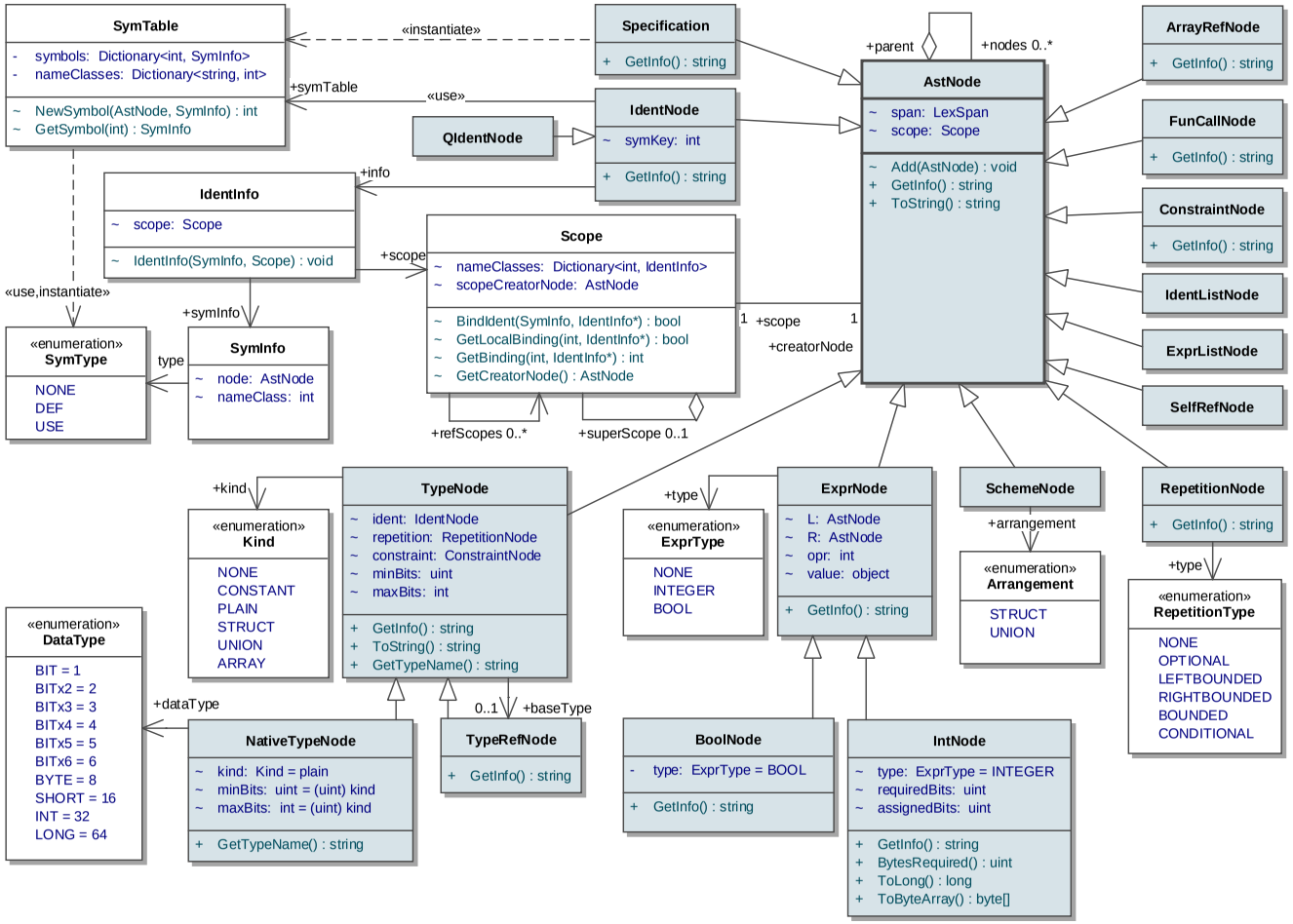


Figure 4.8: *Classes involved in the construction of the syntax tree. The descendants of the AstNode are depicted with a darker background.*

4.3.3 Static Semantic Analysis

As soon as the last node of the syntax tree is created, the analysis of static semantics may be started. For the PBPG language, the main objective in this phase is the construction of a scope hierarchy for name binding and name analysis with regard to this scope hierarchy. Other objectives such as type analysis can also be performed in this phase, however, only to a certain degree, that is, for types of constant structure and size. Because the actually instantiated structure of the types generally depends on runtime values, type analysis in PBPG relates to dynamic semantics and must be done by the binary parsers at runtime. We will focus on that later on.

As explained in clause 2.3.5 the verification of the Algol scope rules requires a two-pass tree traversal: One pass to process all the defining occurrences, and a second one to analyze the applied occurrences. To create an environment of scopes to hold bindings of type definitions, a tree structure is very useful. Based on the *Scope* class from figure 4.8, every *TypeNode* is associated with a new scope, during the first depth-first traversal of the syntax tree. This scope is then used to bind all the subtype definitions. Performed recursively from the root node on, this yields a scope-tree as in figure 4.9.

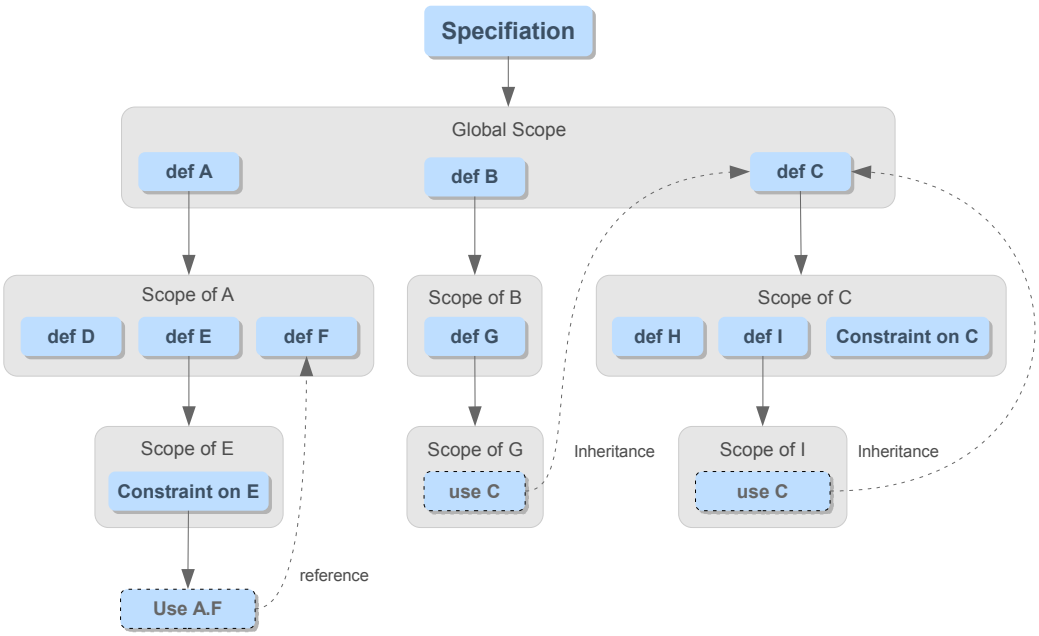


Figure 4.9: *Every type definition creates a new scope within its smallest enclosing scope. Being performed recursively for all types yields the scope-tree.*

In the second depth-first traversal, the applied occurrences of type names must be verified to comply with the scope rule. With regard to the illustrated scope-tree, this rule can be formulated in the following way: The applied occurrence of an identifier is valid whenever it refers to a binding in the local scope or in one of the scopes on the

path towards the root, whereas the nearest binding counts. For the scope-tree above, this means that within the scope of *E* the names *A* through *F* are known, but not the names *G, H, I*. These may be accessed only by means of qualified identifiers starting with the identifier of the accessible types, i.e. *B.G* and *C.H, C.I*. Together with the possibility for type inheritance - which involves jumps through the scope-tree - the name analysis of qualified identifiers is not a trivial task. Hence, the algorithm to perform this is dedicated the own clause 5.2.

An additional traversal of the syntax tree is used to propagate and aggregate bit-sizes of constant composite types and type checks in expressions, ensuring type compatibility with applied operators[18].

4.3.4 Back-end

After successfully passing the analysis steps, the abstract syntax tree is considered to be valid. Since no optimization steps are performed at this moment, the code synthesis represents the last operation in the compilation process. However, future work might introduce optimizations in this phase.

The technical aspect of code synthesis is a matter of streaming textual output into a file. In PBPG, a dedicated *CodeGenerator* class takes care of this task. By traversing the syntax tree once, all the necessary code blocks can be generated and emitted to the output as explained in clause 2.3.6.

However, there exists a difficulty in mapping applied occurrences of type names to correct variable references in terms of correct output program code. The algorithm to do that determines the path from the location of the applied occurrence to the exact scope the referenced identifier is bound to and creates a qualified name representing a reference within the object hierarchy of the generated binary parser. But prior to explain this algorithm in the Implementation clause, the architectural model of the binary parsers will be presented first.

[18] For optimization reasons these calculations can also be shifted to the second pass as well.

4.4 Binary Parser Model

The most interesting and focal part of the design concept is probably the architecture of the binary parsers synthesized by the compiler. Being developed totally from scratch, the binary parser model realizes a form of recursive descent parser we introduced in the foundations. Thus, in analogy to our initial algorithm in figure 2.21, it inherits the idea of mutually recursive parse-function calls to parse sequences, alternatives, repetitions and plain integer values, guided by semantic constraints. Furthermore, the model targets the demand to store the parsed input data in a way such that calculations and further analysis can be performed on it. This regards not only semantic constraint evaluation, but also potential off-line analysis at a later time.

Putting everything together, we can outline that the realized parsing concept is based on a semantic-directed, recursive descent procedure with integrated type instantiation whose final output is a data-object-tree - a syntax tree of binary structures representing the input data.

The remaining questions to answer in order to obtain a functional binary parser are:

- How are the PBPG language constructs, i.e. the types, their structure and the semantic constraints mapped to constructs available in C# and how are those interconnected?
- And, how can the input be read bitwise?

In the following clauses these questions are answered, though in reversed order.

4.4.1 Input Memory Model

The PBPG-generated binary parsers make heavy use of bitwise input reading. This required functionality is offered by a *BitReader* class, which implements a wrapper for the byte-oriented abstract class *Stream* which is part of the C#-framework. *BitReader* allows to read bit strings of arbitrary length, independent from the actual data source implementation derived from *Stream*.

Internally, an input buffer decouples frequent reading accesses by the *BitReader* from potentially slower read procedures of the Stream object handed in. This buffer is provided by another C#-framework class called *BufferedStream*. We depict this configuration and the views of the input data in figure 4.10 and figure 4.11.

The core procedure *ReadBits(int nBits)* of the BitReader relies on an internally maintained bit offset and the most recent byte value. It assures that whenever multi-byte sequences have to be read or bit strings exceeding the number of remaining bits of the recent byte, appropriate chunks of data are fetched from the input buffer. The partitioned data is written to a byte field and returned to the caller.

To allow rewinding the input after mismatching structural patterns during parsing, *BitReader* offers a *Seek* method which controls not only the input offset of the *BufferedStream* object, but also the internal bit offset within the current byte.

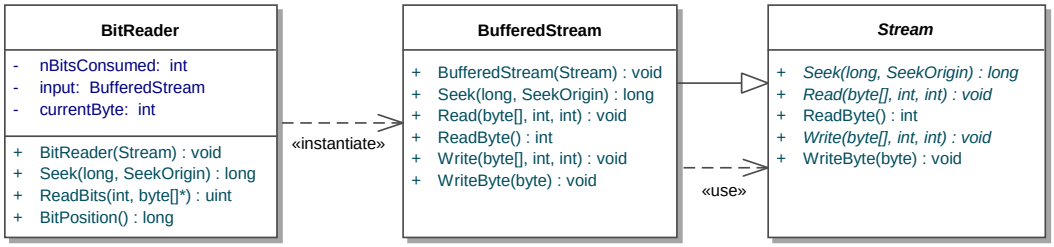


Figure 4.10: *The configuration of the input memory model ensures a decoupling of the potentially slower Stream object by means of an intermediate BufferStream.*

The process of partitioning the input byte stream with regard to the structure specification and the different views of the data are shown in the picture 4.11.

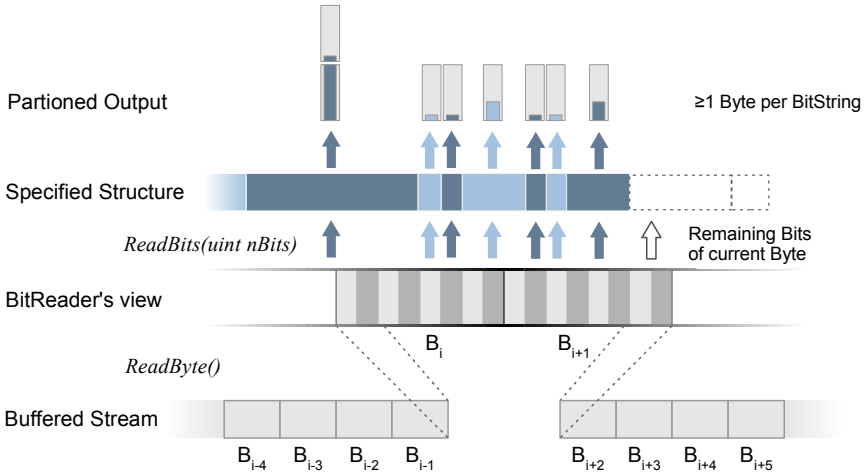


Figure 4.11: *The views of the input data and its mapping to the bit-grained data structures.*

Although not part of the requirements stated for PBPG, the tasks of the *BitReader* could enclose handling of *endianess*, i.e. the byte order within multi-byte data chunks. It is not yet part of the current implementation, however future work might require dynamic changes of endianess during interpretation of integers. Therefore, an additional parameter would control how the data is written to the output byte field, straight or in reversed order.

4.4.2 Language Construct Mapping

The generated C# code reflects the type hierarchy of the PBPG structure specification based on the central *MetaType* class. This class serves as a parameterized descriptor for the data type it represents, storing its bit-size, name and the data gained from parsing. Furthermore, it defines the primitive (non-recursive) parsing method to read raw bit strings of the stored size by means of the *BitReader*. Using *MetaType*, binary parsers for the native data types can be created by correct parameterization of the constructor.

On top of these terminal binary parsers, composite types are defined by nesting. Thus, PBPG structs, unions, and arrays define their subtypes (or element types respectively) as nested classes, preparing variable declarations of those classes as members of the composite type, which is presented in figure 4.12. The parsing method then takes care of calling the parsing methods of the subclass-objects, which are only instantiated at this point in time (*lazy instantiation*). This allows to handle recursive type definitions. In contrast, *eager instantiation* would try to create all required objects through recursive constructor-calls, before the objects are actually required. This would obviously result in a non-terminating instantiation recursion for type definitions like the following:

```
TLV { tag 55; len BYTE; data TLV; };
```

Semantic constraints are translated to Boolean *PostParseCondition* methods of the corresponding classes. They formulate the semantic constraints in terms of expressions over runtime *MetaType* objects to be evaluated right after execution of the local parsing method. Its result dictates the parsing success of its containing class. In case of conditional arrays, the semantic constraints are translated to *ParseCondition* methods, which are evaluated on every iteration of element parsing. As long as the *ParseCondition* returns true, the array accumulates new parsed elements. As soon as the condition evaluates negative, the process stops. Please consider the tables 4.13 and 4.14 for a source code view of the language construct mapping.

The parsing process continues as long all recursive calls return positive, otherwise backtracking is performed in an exhaustive manner. A successful ending then yields a tree of *MetaType* objects filled with the input data. In case of running out of alternatives during back-tracking a partially instantiated object tree remains for error reporting and the root parsing method returns false.

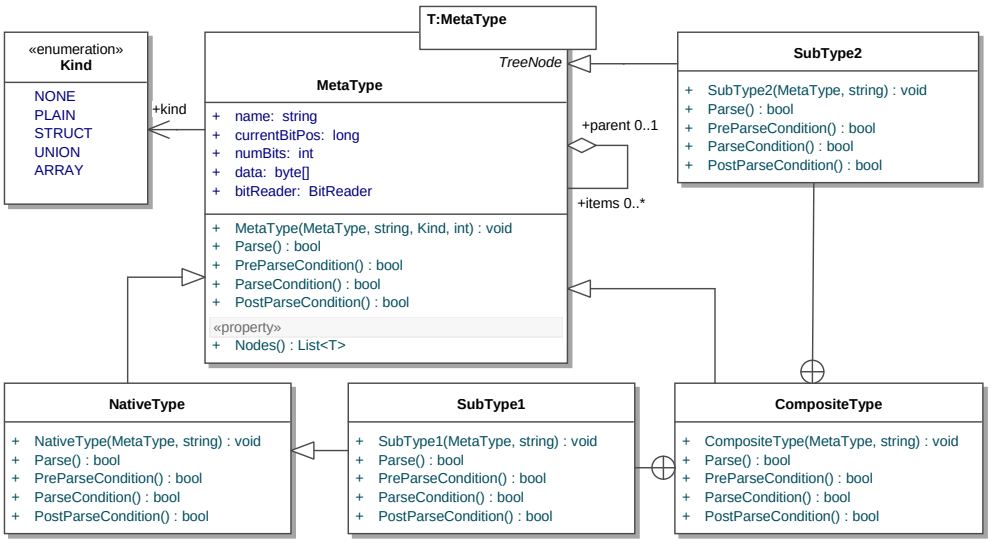


Figure 4.12: *PBPG types are translated to direct or indirect descendants of the MetaType class. Lazy instantiation of these classes during parsing allows recursive type definitions.*

PBPG construct	C# representation
`NewType BaseType;`	`class NewType: BaseType{ ... };`
`StructType { SubType1 SHORT; SubType2 < SubType2a StructType; SubType2b INT; >; };`	`class StructType: MetaType { ... class SubType1: MetaType {...} class SubType2: MetaType { ... class SubType2a: StructType {...} class SubType2b: MetaType {...} } }`
`NewType { a INT :a>255; };`	`class NewType: MetaType { NewType(): base(STRUCT,0) {...}; a m_a; bool Parse() { m_a = new SubType(...); return a.Parse(); } class a: MetaType { a(): base(PLAIN,32){...}; bool Parse() { return (base.Parse() && PostParseCondition()); } bool PostParseCondition() { return this>255; } } }`

Table 4.13: *PBPG specification and corresponding generated C# code. To avoid name collisions between PBPG types and reserved words in C# we use "m_" as a prefix for member field names.*

PBPG construct

```
Str   BYTE[:@!=0];
```

C# representation

```
class Str: MetaType
{
   Str(): base(ARRAY,0) {...}

   List<MetaType> m_StrElements;
   ...
   bool Parse() {
      m_StrElements = new List<MetaType>();
      MetaType newElement;
      do {
         newElement = new MetaType(PLAIN,8);
         if (!newElement.Parse())
            return false;

         m_StrElements.Add(newElement);
      }
      while(ParseCondition(newElement));

      return PostParseCondition();
   }

   bool ParseCondition(MetaType element){
      return (element != 0);
   }
   ...
}
```

PBPG construct

```
IntType   INT;
```

C# representation

```
class IntType: MetaType
{
   ...
   bool Parse() {
      currentBitPos =
         bitReader.BitPosition;

      if (0 == bitReader.ReadBits(
                        maxBits, out data))
      {
         bitReader.Seek(currentBitPos,
            SeekOrigin.Begin);
         return false;
      }
      return PostParseCondition();
   }
   ...
}
```

Table 4.14: *PBPG specification and corresponding generated C# code. To avoid name collisions between PBPG types and reserved words in C# we use "m_" as a prefix for member field names.*

The outcome of successful binary parsing is again an object tree comparable in style with the syntax-tree from figure 4.7 we obtained by parsing specifications. However, this tree consists of *MetaType* objects representing the successfully consumed input data. Based on this tree, convenient data export or further analysis in subsequent steps can be performed. In the next section, the implemented generic method for tree-processing is presented. It provides an initial opportunity to dump the tree to an XML file.

5 Implementation

This section focuses on some noteworthy realization details in the PBPG. It covers the generic and modular concept of tree construction along with visitor-based tree-walkers and the implementation of name analysis, dealing with qualified identifiers and inheritance. Finally, the translation of PBPG name references to correct C# object-names in the binary parser class hierarchy is explained.

5.1 Tree Walking with Visitors

The PBPG compiler, the binary parser model and most parsers in general have in common that they create object trees during parsing for post-processing. In this thesis this commonality is addressed in a generic and modular way. For a better separation of the tree node implementation, the specific analysis code and the tree walking algorithm, a variant of *visitor design pattern* is employed. The design pattern helps in decoupling those three concerns, such that clean development of each individual part is achieved. Furthermore, it improves extendability of all parts.

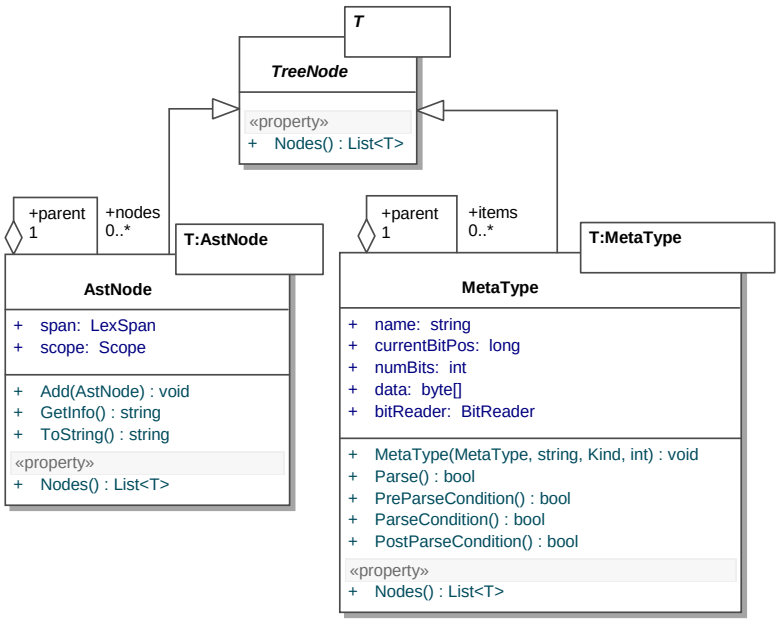


Figure 5.1: *Both the AstNode class of the PBPG compiler and the MetaType class of the binary parser model implement the Nodes property from the abstract generic class TreeNode they derive from.*

First of all, to be traversable by the algorithm, the representatives of the tree nodes, i.e. the *AstNode* and the *MetaType* classes implement the *Nodes*-property from the abstract generic *TreeNode* class as in figure 5.1. This establishes a generic way to access sub nodes of a tree node.

In the second step, the specific analysis code is encapsulated in the tree analyzer classes *ASTAnalyzer* and *MetaTypeAnalyzer*. These inherit from the generic *TreeWalker* class the capability to traverse trees of the specific node type passed in as the generic type parameter.

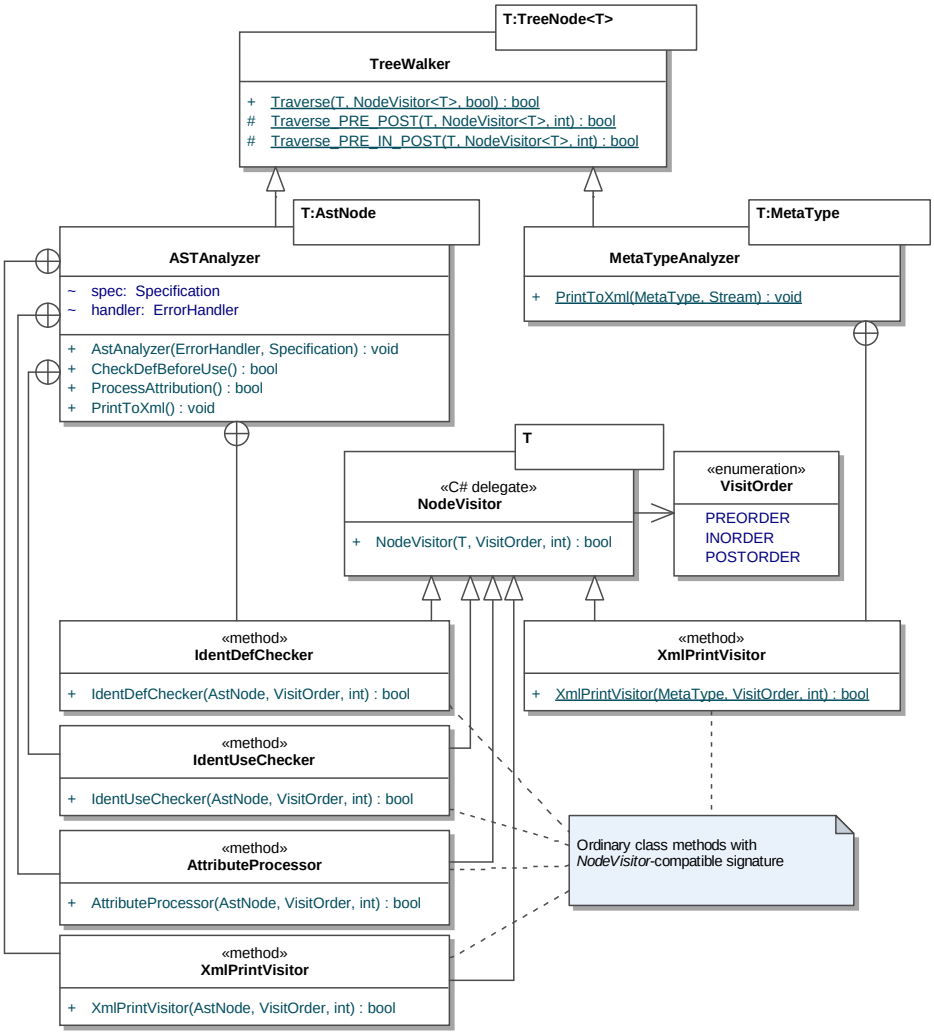


Figure 5.2: *ASTAnalyzer and MetaTypeAnalyzer provide the analyzing visitor methods to be called by the generic TreeWalker implementation during tree traversal.*

The actual tree-walking procedure resides in the *TreeWalker* class itself, whose current implementation performs a depth-first recursive visiting of nodes in pre-order, in-order and post-order. Whereas the analysis code resides in methods of the specific *ASTAnalyzer* and *MetaTypeAnalyzer*, it can be applied on nodes during tree traversal, by passing the methods as a parameter to the inherited *Traverse* operation. Such method passing is enabled by the C#-specific construct *delegate*. It is a *method type* which defines a method signature as a reference for other (compatible) methods. In the example above, all visitor methods realize the signature of the *NodeVisitor* delegate type, and thus can be relayed as parameters to the *Traverse* method.

For better visibility we artificially exposed the visitor methods realizing name analysis, attribute propagation and tree-dumping to XML.

5.2 Analysis and Translation of Name References

As mentioned in clause 4.3.3, the verification of correctly applied occurrences of qualified identifiers in combination with inheritance forms a challenging problem, requiring searches within the abstract syntax tree and the scope tree. Please consider the following PBPG specification, which defines the composite types `A` and `B`.

```
A  {
   a  BYTE;
   b  {
      c  BYTE;
      d  BYTE;
   };
};

B {
   a  A;
   b  BYTE[a.b.c];    //correct
   c  BYTE;
}: a.c != 0;          //incorrect
```

The array type `b` in `B` refers to substructure `b.c` of the local type `a`, which is a descendant of `A`. Additionally, the semantic constraint of `B` refers to the inherited substructure of a. For such constructs verification of applied occurrences requires a combination of tree-traversal and jumps through the syntax-tree whenever inheritance occurs.
The algorithm *FindBinding* verifies the applied occurrences such that inheritance of types is covered. Given an initial scope and the qualified identifier $qid = id\ (.\ id)^*$ to find the binding for, *FindBinding* performs the following steps:

- From the initial scope, traverse the scope-tree towards the root
  1. If there is no scope the first identifier *id* of *qid* is bound to, then the applied occurrence is incorrect.
  2. otherwise, repeat for the subsequent identifiers *id*
     a) Check first if there exists a local binding for *id*
     b) If not, consider inheritance and jump to the scope of the ancestor type to find the binding there.
- If the above iteration failed at some *id* of *qid*, the applied occurrence is considered incorrect.

We next present the pseudocode of *FindBinding* in figure 5.3 and the helper functions it builds on.

Helper Function	Description
SubId(*qid*)	Returns a subsequence omitting the first id or null, *Example:* SubId(a.b.c) yields b.c
GetLocalBinding(*scope, id*)	Returns the *TypeNode* associated to *id*, if *id* is bound to *scope*
GetBinding(*scope, id*)	Returns the *TypeNode* associated to *id*, if *id* is bound to any *scope* on the path towards the root.
BaseTypeId(*typeNode*)	Identifier of the type's ancestor.
SubScope(*typeNode*)	The scope created by typeNode

```
1:  function FINDBINDING(scope, qid): Type
2:      // try to find a binding through the scope tree
3:      TypeNode currentType ← GetBinding(scope, qid);
4:      if (null ≠ currentType) then
5:          subId ← SubId(qid);
6:          // iterate through the elements of id
7:          while (null ≠ subId ∧ null ≠ currentType) do
8:              // is there a local binding?
9:              TypeNode t ← GetLocalBinding(scope, subId);
10:             if (null ≠ t) then
11:                 currentType ← t;
12:             else
13:                 // probably the type is inherited
14:                 TypeNode baseType ← GetBinding(scope, BaseTypeId(currentType));
15:                 if (null ≠ baseType) then
16:                     // then the name of the base type must be known
17:                     currentType ← GetLocalBinding(SubScope(baseType),subId);
18:             // prepare for next iteration
19:             subId ← SubId(subId);
20:     // is there a binding?
21:     if (null = currentType) then
22:         Print("qid is unknown in the current context");
23:     return currentType;
```

Figure 5.3: *Pseudocode of FindBinding for the verification of applied occurrences in PBPG specifications.*

5.2.1 Translation of Name References

For the generated code, the referenced type names from the input specification document must be translated to valid object names in the context of the binary parser class hierarchy. For example, an applied occurrence "*a.b.c*" as used on page 75 must be translated to "*parent.m_a.m_b.m_c*" [19] to be a valid reference of the runtime object *m_c* in the context of the class definition of *b*.

In general, four cases must be distinguished when translating names referenced from within the scope of some arbitrary type *t*. In the examples below some external boolean function `f` is applied to the type denoted by its parameter.

1. The name represents an ancestor of *t* in the type hierarchy

```
A  {
   a  BYTE;
   B  {
      t  BYTE: f(A);      translated to:   parent.parent
   };
};
```

2. The name represents a child of an ancestor of *t*

```
A  {
   a  BYTE;
   B  {
      t  BYTE: f(a);      translated to:   parent.parent.m_a
   };
};
```

3. The name represents *t* itself

```
t  BYTE: f(t);            translated to:   this
```

4. The name represents a child of *t*

```
t  {
   c  BYTE;
}: f(c);                  translated to:   this.m_t
```

For name translations on behalf of code synthesis the *CodeGenerator* class maintains an internal *TranslateNameReference* procedure, where the positive outcome of prior applied occurrence verification is a necessary pre-condition for correct results[20].

[19] *parent* refers to the superior runtime *MetaType* object defined by the enclosing class. See class-diagram in figure 4.12 and the class-nesting concept in table 4.13.

[20] disregarding that compilation is discontinued on analysis-errors anyway

In analogy to *FindBinding*, the *TranslateNameReference* procedure traverses the syntax tree towards the root, however not aiming verification of correctness, but rather in search of the relative path to the actual binding. Please consider that the search is required to obtain the ascending part of the path only, which covers the first three cases from the previous page. The last case represents a descent in the object hierarchy which consists of the actual names of the subtypes already provided by the qualified identifier. In this case, no search is required.

Given an initial scope and the identifier *id* to translate, the algorithm performs the following steps:

1. If the type node which created the initial scope is named *id*,
   print `"this"` and terminate.
2. otherwise print `"parent"` and switch to the superior scope
   a) If the type node which created the scope is named *id*, terminate.
   b) If a sub node of the type node is named *id*,
      print `".m_id"` and terminate.
   c) If the type node represents a derived type,
      switch to scope of the base type and proceed with 2.a)
   d) proceed with 2.

Finally, every *id* in *qid* except the first is printed with a ".m_" prefix, completing the translated name reference.

6 Evaluation

In clause 4.1, we outlined quality requirements to be stated on the designed specification language, its compiler and the generated binary parsers. Based on those requirements, we are now going to evaluate the outcome of the project in the same order.

6.1 Language Quality

Functionality *(Expressiveness, Accuracy, Suitability)*

The PBPG language design covers the structural patterns *sequence*, *alternative* and various forms of *repetitions* frequently used in binary communication protocols. Providing elementary numeric types of common bit sizes 8, 16, 32, 64 and all bit sizes below the byte-level (1 - 7 bit), the language directly addresses the demand of modeling bit-grained data structures. Accordingly, it allows statement of binary numbers by means of a single `#` prefix, which is as simple as for octal and hexadecimal numbers with the prefixes `0` and `$`. Finally, the language supports annotation of types with semantic constraints in terms of a boolean expression over specified structure elements and the opportunity to call externally implemented functions for complex evaluations (e.g. checksum validation). By means of semantic constraints *tagging*, *length-indication* and other indication patterns can be modeled in a simple and intuitive way.

Usability *(Understandability, Convenience)*

As the language syntax is oriented on the C notation of structs, a clean programming style supports spatial emphasis of the specified hierarchies by means of nesting and indentation. Reusing common symbols for structuring and expression operators, the language supports learning and convenience of its user. PBPG narrows the semantic gap we illustrated in figure 4.1 such that the manual part of transcription is reduced to formalization only. Although heavy daily use has to prove it, we think that using PBPG for binary parser construction is much more attractive compared to the manual implementation.

Maintainability *(Analyzability, Changeability)*

The PBPG language is simple and has a small number of syntactic elements, such that the specification documents are easy to read and understand. Since the language encourages reuse of data structures, changes to the specification can be done with less effort compared to a manual implementation. Also, the declarative form of the specification is a clear advantage for changeability. For example, a change in the length of a field might cause a cascaded shifting of all subsequent fields. In manual implementations, this is certain to cause a lot of effort. Although planned, there are no syntactic elements for embedding meta-information, such as references to the normative document or version information at the moment. However, neither the design nor the implementation impede this extension to be introduced as soon as possible.

6.2 Compiler Quality

Functionality *(Correctness)*

For correctly specified data structures, the current implementation produces correct, compilable and functional output code. However, there are situations where a verification of correctness cannot be proven. For example, the existence of an externally defined function used in a semantic constraint can not be verified at the moment. But since the C# compiler does recognize missing references, such errors can be found very quickly. Although construction of test cases for every possible configuration of language constructs can hardly be realized, confidence in the correctness of the compiler can be increased by incrementally adding new test case files to the test suite of the compiler. To the best of our belief, no correct specification cause errors, and most of the incorrect specification produce an error in the output. One of the exceptions is the detection of indirect cyclic inheritance, which is not yet covered by the current analysis of static semantics and is not reported as an error.

Reliability (*Robustness*, *User Support*)

At the moment, a couple of language constructs are not implemented completely. In particular, analysis of static semantics has to be completed in many ways. However, the architecture is prepared for the extension of analysis without significant effort, as the error-processing infrastructure is ready, and the error messages can be emitted with reference to the input document location.

Efficiency (*Time and Memory Consumption*)

The compilation of the PBPG specification file listed in A.3, which specifies UICC instructions of the ETSI TS 102 221 standard, took 87 milliseconds and consumed about 8488 KBytes memory. The test system utilized an Intel Core2 Quad CPU running at 2.83GHz with 4096 MBytes RAM. Although further optimizations of the process are likely to occur in the future, these results are already satisfying.

Maintainability

The architecture of the compiler clearly separates distinct concerns. This is achieved through effective application of design patterns. The compiler combines the generated automata, the error processing module adopted from the construction tools, and the remaining modules discussed in a way such that each of them can be modified or replaced with adequate effort.

Portability (*Retargeting*)

As the generator tools GPLEX and GPPG are compatible to LEX and *yacc*/Bison to a great extend, retargeting of the generated scanner and parser to C or C++ might be achieved with little effort. However, porting the other parts of the compiler from

C# to C would require more work, due to substantial differences in memory handling and language features. Actually, porting the compiler to Java which exhibits similar concepts as C# is probably more easy.

6.3 Quality of Generated Parsers

Functionality (*Process-oriented Parsing*)

At the moment, this requirement is barely addressed in the binary parser architecture. However, modeling data structures in an adequate way can provide the desired behavior. As we hinted by the end of clause 2.4.2 in the context of the prioritized choice, modeling ordered alternatives with decreasing restrictions forms a good strategy to handle unexpected input. We discussed this strategy in clause 3.2 as well. Moreover, if applied on the basis of protocol packets (APDUs) which have a guaranteed ending, every new packet forms a recovery point for a new parsing attempt.

Reliability (*Robustness*)

Compared to the manual implementation of binary parsers, the space for individual programming errors is dramatically narrowed. Being systematically constructed, the dissection procedures are certain to found a stable process with little potential for undiscovered errors. However, care must be taken that memory consumption of the entire process is under control, which might form a problem for large data structures consisting of many bit-sized fields.

At the moment, parsing failure is not reported to the user. However, the concept of a *parse descriptor* as presented in PADS in clause 3.4 hints a promising direction for further development.

Efficiency (*Online Performance*)

After generating the binary parsers from the PBPG specification file from A.3, parsing of a 7 MByte large input file was performed. The file consisted of a repetition of 3 correctly encoded APDU byte strings. The measurements showed that the process finished after about 380 seconds and consumed over 170 MByte of memory prior to completion. These values show that a lot of optimizations in the generated code has to be done to reach acceptable ranges. Although not suited for real life application at the moment, we expect much better results soon.

7 Summary and Future work

In this thesis, we presented the problem of parsing binary data for the purpose of monitoring and analyzing of the communication over the smart card interface as required at COMPRION. Aiming relief of the manual specification transcription required to produce operational parsing logic from informal declarative standard documents we developed the Paderborn Binary Parser Generator - A domain specific language and its compiler for the construction of binary parsers. After highlighting the foundations of smart cards, the protocol monitoring task and aspects of compiler theory we discussed the characteristic differences between textual and binary data with respect to parsing. We showed how the lack of explicit structure in binary inputs is compensated by indication patterns, which guide the parsing process by means of context information. This led us to a modified analysis model incorporating a merge of the lexical and syntactical analysis and a back coupling of the semantic analysis. After proposing a system of types for the realization of a modeling language, we presented the design of the declarative PBPG language and its compiler. We outlined major aspects of the architecture and the difficulties faced during implementation on the background of requirements we found to be important for a high quality result.

Although the concept appeared to solve the problem well and many of the stated requirements have been fulfilled, a lot of conceptual work is remaining for the future. The list of tasks to be faced as soon as possible includes amongst other things the following objectives.

1. Static semantic analysis does not recognize cyclic inheritance in PBPG types. This verification has to be added in order to ensure seamless compilation of generated C# code. Furthermore, the verification of expressions with regard to type compatibility has to be faced.
2. The currently supported form of array definition we listed in clause 4.2 is *constant* i.e. `ArrType ElementType[m]`. Support of conditional and bounded arrays will provide the aspired convenience in modeling. Especially the conditional array definition is a powerful and demanded construct to reflect constraints over repetitions incorporating internal order or terminal symbols, e.g. null-terminated strings. Moreover, the automated recognition of ambiguity as outlined in clause 4.2 should be considered.
3. Implementation of an inclusion system for reuse of existing PBPG specification files and an opportunity to bind to existing generated binary parsers.
4. Inclusion of meta-information such as normative document references and original names (which sometimes consist of multiple words) into the specification document.
5. At the moment, the generated binary parser code is not optimized, which means that unnecessary methods and declarations exist, e.g. every class in the *MetaType* class hierarchy implements its own *PostParseCondition* method, even if no semantic constraint is assigned to the corresponding PBPG type. This enlarges

the source code files unnecessarily and is uncertain to be optimized by the C# compiler under all circumstances.

6. In accordance to PADS in clause 3.4 and binpac in clause 3.5 the parameterization of types might be worth considering, as it offers an additional dimension of modularity to structure specifications and has potentially a similar importance as templates in Java or generics in C#. However, the realization might probably bring significant modifications to the overall compiler architecture.
7. On the long run, we can imagine that formalized structure specifications can be used to analyze consistency of the informal normative documents in order to unveil incorrect formulations by means of automated semantic analysis. Inconsistent statements from informal descriptions might become detectable after formalization, or even immediately obvious during the formalization process.

A Annex

In this section we assembled specification listings mentioned in the document.

A.1 ISO/IEC 7816-4 smart card commands

- ERASE BINARY
- VERIFY
- MANAGE CHANNEL
- EXTERNAL AUTHENTICATE
- GET CHALLENGE
- INTERNAL AUTHENTICATE
- SELECT FILE
- READ BINARY
- READ RECORD(S)
- GET RESPONSE
- ENVELOPE
- GET DATA
- WRITE BINARY
- WRITE RECORD
- UPDATE BINARY
- PUT DATA
- UPDATE DATA
- APPEND RECORD

A.2 PBPG Language Specification

A.2.1 and A.2.2 contain copies of the language specification files used for the construction of the PBPG Scanner and Parser.

A.2.1 Scanner Specification

The listing specifies the PBPG scanner specification for the scanner generator GPLEX.

```
//======================================================================================
//== DEFINITIONS SECTION ==============================================================
//======================================================================================

%using PBPG.Parser;

%namespace PBPG.Lexers

%option stack, verbose, summary, noEmbedBuffers out:Scanner.cs
%visibility internal

eol          (\r\n?|\n)
NonWh        [^ \t\r\n]
Space        [ \t]
DecDig       [0-9]
OctDig       [0-7]
HexDig       [0-9a-fA-F]

NonLE        [^\r\n]
EscChr       \\{NonLE}

Ident        [a-zA-Z_][a-zA-Z0-9_]*
Bin          #[0,1]+
Oct          0{OctDig}+
Dec          {DecDig}+
Hex          ${HexDig}+

StrChr        [^\\\"\a\b\f\n\r\t\v\0]
ChrChr        [^\\'\a\b\f\n\r\t\v\0]

SLC          \/\/        // Single Line Comment
MLC_         \/\*        // Start of Multi Line Comment
_MLC         \*\/        // End of MultiLine Commnet
Line         [^\n\r]*    // Anything but Line End
NonStar      ([^\*]|\*[^\/])*        // Anything but */

StrLit     \"({StrChr}|{EscChr})*\"
ChrLit     \'({ChrChr}|{EscChr})*\'

//======================================================================================
%% //== RULES SECTION ===================================================================
//======================================================================================

{eol}                    {/* SKIP */}
{SLC}{Line}              {/* SKIP */}
{MLC_}{NonStar}{_MLC}    {/* SKIP */}

{StrLit}                 {yylval.sString = yytext; return (int)Tokens.STRLIT;}
{ChrLit}                 {yylval.sString = yytext; return (int)Tokens.CHRLIT;}

".."                     {return (int)Tokens.BIDOT;}
"."                      {return (int)Tokens.DOT;}
","                      {return (int)Tokens.COMMA;}
";"+                     {return (int)Tokens.SEMI;}
```

```
":"                     {return (int)Tokens.COLON;}
"?"                     {return (int)Tokens.QUESTION;}
"@@"                    {return (int)Tokens.BIATSIGN;}
"@"                     {return (int)Tokens.ATSIGN;}
"^"                     {yylval.tkn = Tokens.CARET;     return (int)yylval.tkn;}
"||"                    {yylval.tkn = Tokens.BIPIPE;    return (int)yylval.tkn;}
"|"                     {yylval.tkn = Tokens.PIPE;      return (int)yylval.tkn;}
"&&"                    {yylval.tkn = Tokens.BIAMP;     return (int)yylval.tkn;}
"&"                     {yylval.tkn = Tokens.AMP;       return (int)yylval.tkn;}
"=="                    {yylval.tkn = Tokens.EQ;        return (int)yylval.tkn;}
"!="                    {yylval.tkn = Tokens.NEQ;       return (int)yylval.tkn;}
"!"                     {yylval.tkn = Tokens.EXCL;      return (int)yylval.tkn;}
"<<"                    {yylval.tkn = Tokens.BILT;      return (int)yylval.tkn;}
"<="                    {yylval.tkn = Tokens.LTE;       return (int)yylval.tkn;}
"<"                     {yylval.tkn = Tokens.LT;        return (int)yylval.tkn;}
">>"                    {yylval.tkn = Tokens.BIGT;      return (int)yylval.tkn;}
">="                    {yylval.tkn = Tokens.GTE;       return (int)yylval.tkn;}
">"                     {yylval.tkn = Tokens.GT;        return (int)yylval.tkn;}
"++"                    {Error(10, TokenSpan());        return (int)Tokens.TOKERR;}
"+"                     {yylval.tkn = Tokens.PLUS;      return (int)yylval.tkn;}
"--"                    {Error(10, TokenSpan());        return (int)Tokens.TOKERR;}
"-"                     {yylval.tkn = Tokens.MINUS;     return (int)yylval.tkn;}
"*"                     {yylval.tkn = Tokens.STAR;      return (int)yylval.tkn;}
"/"                     {yylval.tkn = Tokens.SLASH;     return (int)yylval.tkn;}
"%"                     {yylval.tkn = Tokens.PC;        return (int)yylval.tkn;}

"("                     {return (int)Tokens.LPAREN;}
")"                     {return (int)Tokens.RPAREN;}
"["                     {return (int)Tokens.LBRACK;}
"]"                     {return (int)Tokens.RBRACK;}
"{"                     {return (int)Tokens.LBRACE;}
"}"                     {return (int)Tokens.RBRACE;}

{Ident}                 {
                            yylval.sIdent = yytext;
                            return (int)GetKeyword(yytext);
                        }

{Bin}                   {
                            yylval.iVal = ParseNum(yytext.Substring(1), 2);
                            return (int)Tokens.INTEGER;
                        }

{Oct}                   {
                            char peek = (char)Peek();

                            if(peek != '8' && peek!='9')
                            {
                                yylval.iVal = ParseNum(yytext.Substring(1), 8);
                                return (int)Tokens.INTEGER;
                            }
                        }
{Dec}                   {
                            yylval.iVal = ParseNum(yytext, 10);
                            return (int)Tokens.INTEGER;
                        }
{Hex}                   {
                            yylval.iVal = ParseNum(yytext.Substring(1), 16);
                            return (int)Tokens.INTEGER;
                        }

<*>{NonWh}              {Error(1, TokenSpan()); } /* Unknown symbol in this context */

<<EOF>>                 {return (int)Tokens.EOF;}
```

```
%{
    // Scanner epilog
    yylloc = new LexSpan(tokLin, tokCol, tokELin, tokECol, tokPos, tokEPos, buffer);
%}

//==========================================================================================
%% // == USER-CODE SECTION empty ============================================================
//==========================================================================================
```

A.2.2 Parser Specification

The listing specifies the PBPG language grammar for the parser generator GPPG.

```
//==========================================================================
//== DEFINITIONS SECTION ==================================================
//==========================================================================

%output=Parser.cs

%namespace PBPG.Parser

%YYLTYPE LexSpan
%partial
%visibility internal


%union {
    public long iVal;
    public bool bVal;
    public Tokens tkn;
    public string sIdent;
    public string sString;
    public AstNode node;
    public TypeNode typeNode;
    public List<AstNode> nodeList;
}

%start Unit

%token <iVal> INTEGER "Integer value"
%token <bVal> BOOL "Bool value"
%token <sIdent> IDENT "Identifier"
%token BIDOT "..", DOT ".", COMMA ",", SEMI ";", COLON ":", QUESTION "?"
%token LPAREN "(", RPAREN ")", LBRACK "[", RBRACK "]", LBRACE "{", RBRACE "}"
%token kwUNIT "unit", kwUSES "uses", kwIMPORT "import", kwFROM "from"
%token EOF "end of file", TOKERR

//== String literals ======================================================
%token <sString> STRLIT CHRLIT

//== Buil-in Types ========================================================
%token ntBITx2 "2BIT" ntBITx3 "3BIT" ntBITx4 "4BIT"
%token ntBITx5 "5BIT" ntBITx6 "6BIT" ntBITx7 "7BIT"
%token ntBIT "BIT" ntBYTE "BYTE" ntSHORT "SHORT"
%token ntINT "INT" ntLONG "LONG"

//== Left associative operators ===========================================
%left ATSIGN "@", BIATSIGN "@@"
%left <tkn> CARET "^"
%left <tkn> PIPE "|"
%left <tkn> BIPIPE "||"
%left <tkn> AMP "&"
%left <tkn> BIAMP "&&"
%left <tkn> EQ "=="
%left <tkn> NEQ "!="
%left <tkn> LT "<"
%left <tkn> LTE "<="
%left <tkn> GT ">"
%left <tkn> GTE ">="
%left <tkn> BILT "<<"
%left <tkn> BIGT ">>"
%left <tkn> PLUS "+"
%left <tkn> MINUS "-"
%left <tkn> STAR "*"
```

```
%left <tkn> SLASH "/"
%left <tkn> PC "%"

%token <tkn> NOP

//== Right associative operators ==========================================
%right <tkn> EXCL "!"

//== Nonterminal values ====================================================
%type <nodeList> TypeDecls
%type <typeNode> TypeDecl Scheme

%type <node>    Unit UnitHdr
%type <node>    SubScheme Repetition Constraint
%type <node>    Block Expr OpLogicOR OpLogicAND OpBinOR OpBinXOR OpBinAND
%type <node>    OpEquate OpCompare OpShift OpAdd OpMult OpUnary
%type <node>    Operand QRef QRefHead FunCall FunCallTail ArrayRef ExprList IdentList
%type <node>    QIdent BaseType Int Bool

%type <tkn>    Eq_Neq L_G Shift_LR Plus_Minus Mul_Div_Pc Neg_Not

//=============================================================================
%% //== RULES SECTION ========================================================
//=============================================================================
Unit        : UnitHdr Includes Imports TypeDecls    {$$ = new UnitNode(
                                                          $1 as IdentNode);
                                                    ($$ as UnitNode).AddTypeDecls($4);
                                                     spec.Add($$);}
            ;

UnitHdr     : kwUNIT IDENT SEMI                     {$$ = new IdentNode(@2, false);}
            ;

Includes    : kwUSES IdentList SEMI                 // NOT YET HANDELED
            |
            ;

Imports     : kwIMPORT QIdent kwFROM STRLIT SEMI    // NOT YET HANDELED
            |
            ;

// ATTENTION: Right recursion can produce stack overflow!!!
QIdent      : IDENT DOT QIdent                      {$$ = new QIdentNode(@1);
                                                     $$.Add($3);}
            | IDENT                                 {$$ = new QIdentNode(@1);}
            ;

TypeDecls   : TypeDecls TypeDecl                    {$$ = $1; $$.Add($2);}
            | TypeDecl                              {$$ = new List<AstNode>();
                                                     $$.Add($1);}
            ;

// NON-ARRAY TYPE:  TypeDecl := Scheme
TypeDecl    : IDENT Scheme Constraint SEMI          {$$ = $2;
                                                     $$.SetIdent(new IdentNode(@1,
                                                            true));
                                                     $$.SetConstr(
                                                            $3 as ConstraintNode);}
            ;

// ARRAY TYPE:  TypeDecl := ArrayType of Scheme Items
TypeDecl    : IDENT Scheme Repetition Constraint SEMI{$$ = new TypeNode(@1,
                                                            $2 as TypeNode,
                                                            $3 as RepetitionNode);
                                                      $$.SetIdent(new IdentNode(@1,
```

```
                                                        true));
                                                    $$.SetConstr(
                                                        $4 as ConstraintNode);}
            ;

Scheme      : BaseType SubScheme                    {$$ = new TypeNode(@1,
                                                        $1 as TypeNode,
                                                        $2 as SchemeNode);}
            | BaseType                              {$$ = new TypeNode(@1,
                                                        $1 as TypeNode,
                                                        null as SchemeNode);}
            | SubScheme                             {$$ = new TypeNode(@1, null,
                                                        $1 as SchemeNode);}
            | Int                                   {$$ = new TypeNode($1 as IntNode);}
            ;

BaseType    : ntBIT                                 {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BIT);}
            | ntBITx2                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BITx2);}
            | ntBITx3                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BITx3);}
            | ntBITx4                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BITx4);}
            | ntBITx5                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BITx5);}
            | ntBITx6                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BITx6);}
            | ntBITx7                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BITx7);}
            | ntBYTE                                {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.BYTE);}
            | ntSHORT                               {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.SHORT);}
            | ntINT                                 {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.INT);}
            | ntLONG                                {$$ = new NativeTypeNode(@1,
                                                        NativeTypeNode.DataType.LONG);}
            | IDENT                                 {$$ = new TypeRefNode(
                                                            new IdentNode(@1, false));}
            ;

SubScheme   : LT TypeDecls GT                       {$$ = new SchemeNode(@2,
                                                        SchemeNode.Arrangement.UNION,
                                                            $2);}
            | LBRACE TypeDecls RBRACE               {$$ = new SchemeNode(@2,
                                                        SchemeNode.Arrangement.STRUCT,
                                                        $2);}
            ;

Repetition  : LBRACK Expr BIDOT Expr RBRACK         {$$ = new RepetitionNode(@1, $2,
                                                            $4);}
            | LBRACK Expr BIDOT RBRACK              {$$ = new RepetitionNode(@1, $2,
                                                            null);}
            | LBRACK Expr RBRACK                    {$$ = new RepetitionNode(@1,
                                                            null, $2);}
            | LBRACK COLON Expr RBRACK              {$$ = new RepetitionNode(@1,
                                                            $3);}
            | LBRACK RBRACK                         {$$ = new RepetitionNode(@1,
                                                            null, null);}
            ;

Constraint  : COLON Block                           {$$ = new ConstraintNode(@2);
                                                     $$.Add($2);}
            |
```

```
            ;

Block       : LBRACE Expr RBRACE                    {$$ = $2;}
            | Expr                                  {$$ = $1;}
            ;

//=============================================================================
//== Expressions ==============================================================
//=============================================================================
Expr        : OpLogicOR                             {$$ = $1;}
            ;

OpLogicOR   : OpLogicOR BIPIPE OpLogicAND           {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpLogicAND                            {$$ = $1;}
            ;

OpLogicAND  : OpLogicAND BIAMP OpBinOR              {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpBinOR                               {$$ = $1;}
            ;

OpBinOR     : OpBinOR PIPE OpBinXOR                 {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpBinXOR                              {$$ = $1;}
            ;

OpBinXOR    : OpBinXOR CARET OpBinAND               {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpBinAND                              {$$ = $1;}
            ;

OpBinAND    : OpBinAND AMP OpEquate                 {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpEquate                              {$$ = $1;}
            ;

OpEquate    : OpEquate Eq_Neq OpCompare             {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpCompare                             {$$ = $1;}
            ;

OpCompare   : OpCompare L_G OpShift                 {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpShift                               {$$ = $1;}
            ;

OpShift     : OpShift Shift_LR OpAdd                {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpAdd                                 {$$ = $1;}
            ;

OpAdd       : OpAdd Plus_Minus OpMult               {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpMult                                {$$ = $1;}
            ;

OpMult      : OpMult Mul_Div_Pc OpUnary             {$$ = new ExprNode(@2,$2,$1,$3);}
            | OpUnary                               {$$ = $1;}
            ;

OpUnary     : Neg_Not OpUnary                       {$$ = new ExprNode(@1,$1,null,$2);}
            | Operand                               {$$ = $1;}
            | LPAREN Expr RPAREN                    {$$ = $2;}
            ;

//=============================================================================
Eq_Neq      : EQ                                    {$$ = $1;}
            | NEQ                                   {$$ = $1;}
            ;

L_G         : LT                                    {$$ = $1;}
            | LTE                                   {$$ = $1;}
            | GT                                    {$$ = $1;}
```

```
            | GTE                                   {$$ = $1;}
            ;

Shift_LR    : BILT                                  {$$ = $1;}
            | BIGT                                  {$$ = $1;}
            ;

Plus_Minus  : PLUS                                  {$$ = $1;}
            | MINUS                                 {$$ = $1;}
            ;

Mul_Div_Pc  : STAR                                  {$$ = $1;}
            | SLASH                                 {$$ = $1;}
            | PC                                    {$$ = $1;}
            ;

Neg_Not     : MINUS                                 {$$ = $1;}
            | EXCL                                  {$$ = $1;}
            ;

//=============================================================================
//== Identifier, FunctionCall, ArrayRef ======================================
//=============================================================================
Operand     : FunCall                               {$$ = $1;}
            | Int                                   {$$ = $1;}
            | Bool                                  {$$ = $1;}
            | QRef                                  {$$ = $1;}
            | BIATSIGN                              {$$ = new SelfRefNode(@1);}
            | ATSIGN                                {$$ = new SelfRefNode(@1);}
            ;

// ATTENTION: Right recursion can produce stack overflow!!!
QRef        : QRefHead DOT QRef                     {$$ = new QIdentNode(@1);
                                                     $$.Add($3);}
            | QRefHead                              {$$ = new QIdentNode(@1);}
            ;


QRefHead    : ArrayRef                              {$$ = $1;}
            | IDENT                                 {$$ = new QIdentNode(@1);}
            ;

FunCall     : IDENT LPAREN FunCallTail              {$$ = new FunCallNode(@1);
                                                     $$.Add($3);}
            ;

FunCallTail : ExprList RPAREN                       {$$ = $1;}
            | RPAREN
            ;

ArrayRef    : IDENT LBRACK Expr RBRACK              {$$ = new ArrayRefNode(@1);
                                                     $$.Add($3);}
            ;

ExprList    : ExprList COMMA Expr                   {$$ = $1; $$.Add($3);}
            | Expr                                  {$$ = new ExprListNode(@1);
                                                     $$.Add($1);}
            ;

Int         : INTEGER                               {$$ = new IntNode(@1, $1);}
            ;

Bool        : BOOL                                  {$$ = new BoolNode(@1, $1);}
            ;
```

```
IdentList    : IdentList COMMA IDENT                {$$ = $1;
                                                     $$.Add(new IdentNode(
                                                                  @3,false));}
             | IDENT                                {$$ = new IdentListNode(@1);
                                                     $$.Add(new IdentNode(
                                                                  @1, false));}
             ;
//=========================================================================================
%% // == USER-CODE SECTION empty ========================================================
//=========================================================================================
```

A.3 UICC Commands formalized in PBPG syntax

The following listing shows an initial rudimentary structure specification of all UICC instructions of the ETSI TS 102221 [Eur09] standard. Based on the definitions of this listing the individual substructure of the instruction bodies has to be specified to obtain binary parsers for the ETSI TS 102221 standard.

```
unit UICC_COMMANDS;

//== UICC COMMAND ===================================================================

COMMAND
<
    group1  CLA_0X_4X_6X_COMMAND;
    group2  CLA_80_COMMAND;
    group3  CLA_8X_CX_EX_COMMAND;
>;

//== INSTRUCTION CLASSES ============================================================

CLA_0X_4X_6X_COMMAND
{
    cla BYTE:   (@ >> 4) == $0 ||
                (@ >> 4) == $4 ||
                (@ >> 4) == $6;

    Instruction
    <
        select_file             SELECT_FILE;
        read_binary             READ_BINARY;
        update_binary           UPDATE_BINARY;
        read_record             READ_RECORD;
        update_record           UPDATE_RECORD;
        search_Record           SEARCH_RECORD;
        verify                  VERIFY;
        change_pin              CHANGE_PIN;
        disable_pin             DISABLE_PIN;
        enable_pin              ENABLE_PIN;
        unblock_pin             UNBLOCK_PIN;
        deactivate_file         DEACTIVATE_FILE;
        activate_file           ACTIVATE_FILE;
        authenticate            AUTHENTICATE;
        get_challenge           GET_CHALLENGE;
        manage_channel          MANAGE_CHANNEL;
        manage_secure_channel   MANAGE_SECURE_CHANNEL;
        transact_data           TRANSACT_DATA;
    >;
};

CLA_8X_CX_EX_COMMAND
{
    cla BYTE:   (@ >> 4) == $8 ||
                (@ >> 4) == $C ||
                (@ >> 4) == $E;

    Instruction
    <
        status                  STATUS;
        increase                INCREASE;
        retrieve_data           RETRIEVE_DATA;
        set_data                SET_DATA;
        terminal_capability     TERMINAL_CAPABILITY;
    >;
};
```

```
CLA_80_COMMAND
{
    cla BYTE:   @ == $80;

    Instruction
    <
        terminal_profile        TERMINAL_PROFILE;
        envelope                ENVELOPE;
        fetch                   FETCH;
        terminal_response       TERMINAL_RESPONSE;
    >;
};


INS_BODY
{
    p1      BYTE;
    p2      BYTE;
    le      BYTE;
    data    BYTE[le];
};

STATUS_WORD
{
    sw1     BYTE;
    sw2     BYTE;
};

//== INSTRUCTIONS ===================================================================

SELECT_FILE             {ins $A4; body INS_BODY; sw STATUS_WORD; };
STATUS                  {ins $F2; body INS_BODY; sw STATUS_WORD; };
READ_BINARY             {ins $B0; body INS_BODY; sw STATUS_WORD; };
UPDATE_BINARY           {ins $D6; body INS_BODY; sw STATUS_WORD; };
READ_RECORD             {ins $B2; body INS_BODY; sw STATUS_WORD; };
UPDATE_RECORD           {ins $DC; body INS_BODY; sw STATUS_WORD; };
SEARCH_RECORD           {ins $A2; body INS_BODY; sw STATUS_WORD; };
INCREASE                {ins $32; body INS_BODY; sw STATUS_WORD; };
RETRIEVE_DATA           {ins $CB; body INS_BODY; sw STATUS_WORD; };
SET_DATA                {ins $DB; body INS_BODY; sw STATUS_WORD; };
VERIFY                  {ins $20; body INS_BODY; sw STATUS_WORD; };
CHANGE_PIN              {ins $24; body INS_BODY; sw STATUS_WORD; };
DISABLE_PIN             {ins $26; body INS_BODY; sw STATUS_WORD; };
ENABLE_PIN              {ins $28; body INS_BODY; sw STATUS_WORD; };
UNBLOCK_PIN             {ins $2C; body INS_BODY; sw STATUS_WORD; };
DEACTIVATE_FILE         {ins $04; body INS_BODY; sw STATUS_WORD; };
ACTIVATE_FILE           {ins $44; body INS_BODY; sw STATUS_WORD; };
AUTHENTICATE            {ins BYTE: @==$88 || @==$89; body INS_BODY; sw STATUS_WORD;};
GET_CHALLENGE           {ins $84; body INS_BODY; sw STATUS_WORD; };
TERMINAL_CAPABILITY     {ins $AA; body INS_BODY; sw STATUS_WORD; };
TERMINAL_PROFILE        {ins $10; body INS_BODY; sw STATUS_WORD; };
ENVELOPE                {ins $C2; body INS_BODY; sw STATUS_WORD; };
FETCH                   {ins $12; body INS_BODY; sw STATUS_WORD; };
TERMINAL_RESPONSE       {ins $14; body INS_BODY; sw STATUS_WORD; };
MANAGE_CHANNEL          {ins $70; body INS_BODY; sw STATUS_WORD; };
MANAGE_SECURE_CHANNEL   {ins $73; body INS_BODY; sw STATUS_WORD; };
TRANSACT_DATA           {ins $75; body INS_BODY; sw STATUS_WORD; };
GET_RESPONSE            {ins $C0; body INS_BODY; sw STATUS_WORD; };
```

Acronyms

3GPP Third Generation Partnership Project
AID Application Identifier
ADF Application Directory File
APDU Application Protocol Data Unit
AG Attribute Grammar
BNF Backus-Naur Form
C-APDU Command APDU
C-TPDU Command TPDU
CDMA Code Division Multiple Access
CFG Context-Free Grammar
DF Dedicated File
EBNF Extended BNF
EF Elementary File
ELF Executable and Linkable Format
ETSI European Telecommunications Standards Institute
FID File Identifier
FSA Finite State Automaton
GPLEX Gardens Point Scanner Generator
GPPG Gardens Point Parser Generator
GSM Global System for Mobile Communications
ISO International Organization for Standardization
IEC International Electrotechnical Commission
MF Master File
MPPG Managed Package Parser Generator
MPLEX Managed Package Scanner Generator
NIDS Network Intrusion Detection System
OSI Open System Interconnection
PBPG Paderborn Binary Parser Generator
PDA Pushdown automaton
PEG Parsing Expression Grammar
PIN Personal Identification Number
R-APDU Response APDU
R-TPDU Response TPDU
SIM Subscriber Identity Module
TPDU Transfer Protocol Data Unit
UICC Universal Integrated Circuit Card
USB Universal Serial Bus
XML Extensible Markup Language

References

[ALSU06] AHO, ALFRED V., MONICA S. LAM, RAVI SETHI, and JEFFREY D. ULLMAN: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 8 2006.

[Bac02] BACK, GODMAR: *DataScript - a specification and scripting language for binary data*. In *GPCE*, pages 66–77, 2002.

[BBW+07] BORISOV, NIKITA, DAVID BRUMLEY, HELEN J. WANG, JOHN DUNAGAN, PALLAVI JOSHI, and CHUANXIONG GUO: *Generic application-level protocol analyzer and its language*. In *NDSS*, 2007.

[CT03] COOPER, KEITH and LINDA TORCZON: *Engineering a Compiler*. Morgan Kaufmann, 1 edition, 11 2003.

[ETS95] ETSI: *Digital cellular telecommunications system (phase 2+); specification of the subscriber identity module - mobile equipment (sim - me) interface (gsm 11.11)*, 1995.

[Eur09] *ETSI TS 102 221: Smart cards: Uicc-terminal interface; physical and logical characteristics (release 8)*, 2009.

[FG05] FISHER, KATHLEEN and ROBERT GRUBER: *PADS: a domain-specific language for processing ad hoc data*. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 295–304, New York, NY, USA, 2005. ACM.

[FMW06] FISHER, KATHLEEN, YITZHAK MANDELBAUM, and DAVID WALKER: *The next 700 data description languages*. SIGPLAN Not., 41(1):2–15, 2006.

[For04] FORD, BRYAN: *Parsing Expression Grammars: A recognition-based syntactic foundation*. In *Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, 2004.

[Gou09a] GOUGH, JOHN: *The gplex scanner generator*. http://gplex.codeplex.com/, 2009.

[Gou09b] GOUGH, JOHN: *The gppg parser generator*. http://gppg.codeplex.com/, 2009.

[ISOa] *ISO/IEC 7816-2: Identification cards – integrated circuit cards – part 2: Cards with contacts – dimensions and location of the contacts*.

[ISOb] *ISO/IEC 7816-3: Identification cards – integrated circuit cards – part 3: Cards with contacts – electrical interface and transmission protocols*.

[ISOc] *ISO/IEC 7816-4: Identification cards – integrated circuit cards – part 3: Cards with contacts – interindustry commands for interchange*.

[ISO94] *ISO/IEC 7498-1:1994: Information technology – open systems interconnection – basic reference model: The basic model*, 1994.

[ISO01] *ISO/IEC 9126-1: Software engineering – product quality – part 1: Quality model*, 2001.

[ISO03a] *ISO/IEC 9126-2: Software engineering – product quality – part 2: External metrics*, 2003.

[ISO03b] *ISO/IEC 9126-3: Software engineering – product quality – part 3: Internal metrics*, 2003.

[ISO04] *ISO/IEC 9126-4: Software engineering – product quality – part 4: Quality in use metrics*, 2004.

[JS00] J., MCCANN PETER and CHANDRA SATISH: *Packet Types: Abstract specification of network protocol messages*. SIGCOMM Comput. Commun. Rev., 30(4):321–333, 2000.

[Kas90] KASTENS, UWE: *Übersetzerbau*. Oldenbourg, 1990.

[Kas09a] KASTENS, UWE: *Grundlagen der Programmiersprachen*. http://ag-kastens.uni-paderborn.de/lehre/material/gps/, 2009.

[Kas09b] KASTENS, UWE: *Programming Languages and Compilers*. http://ag-kastens.uni-paderborn.de/lehre/material/plac/, 2009.

[Knu68] KNUTH, DONALD E.: *Semantics of context-free languages*. Mathematical Systems Theory, 2(2):127–145, 1968.

[LJ09] LEVINE, JOHN and LEVINE JOHN: *Flex & Bison*. O'Reilly Media, 1 edition, 7 2009.

[PPSP06] PANG, RUOMING, VERN PAXSON, ROBIN SOMMER, and LARRY PETERSON: *binpac: a yacc for writing application protocol parsers*. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300, New York, NY, USA, 2006. ACM.

[RE02] RANKL, WOLFGANG und WOLFGANG EFFING: *Handbuch der Chipkarten: Aufbau - Funktionsweise - Einsatz von Smart Cards*. Carl Hanser, 4., überarb. und akt. Auflage, 2002.

[RE03] RANKL, W. and W. EFFING: *Smart Card Handbook*. John Wiley & Sons, 3rd edition, 2003.

[SK95] SLONNEGER, KENNETH and BARRY L. KURTZ: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison Wesley, 1st edition, 9 1995.

[Wir96] WIRTH, NIKLAUS: *Compiler Construction (International Computer Science Series)*. Addison-Wesley, pap/dsk edition, 6 1996.

[Wir05] WIRTH, NIKLAUS: *Compiler construction, revised version of [Wir96]*, 2005.