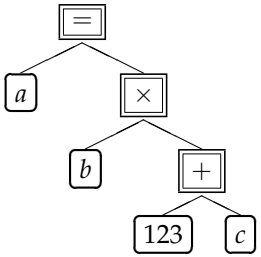
Chapter 3

Language M for describing a measurement procedure

MEASUREMENT procedures are described very precisely by numerous standard operation procedures (SOP) and validation studies as mentioned in Section 1.2. But before a computer system is able to process and analyze a measurement procedure in any way the latter must be described in a *computer readable language*. A list of instructions for the laboratory personnel is not enough in this respect. A computer can give assistance to the tedious work of evaluating and computing the result of a measurement and its measurement uncertainty only if the user translates the SOP into computer readable form. What could such a measurement procedure description look like? A considerable part of this thesis was the development of a language called M with which one can conveniently describe a measurement procedure to the MUSAC-system. It is described in details in Section 3.2. Nowadays, there exist various general purpose description languages and if the MUSAC-project were launched today, we probably would try to use one of those. But when the MUSAC-project started description languages like XML were just about to leave the state of prototype. However, the language M is — in our opinion — particularly nice and well apt for this special purpose. Apart from that, writing your own language is fun!

Figure 7 Syntax Tree for $a = b \times (123 + c)$



3.1 Compiling and Interpreting

A compiler is a program that reads a description of a program written in one language, the *source* language, and translates it into an equivalent description in another language, the *target* language. Why would one not write directly in the target language? Because the final target language for computer programs is a long sequence of bits. Nobody would be interested in manipulating these very long sequences of zeros and ones "manually". Compiler and interpreter reliably perform this very tedious and error prone task of translating a program from a high level language into finally executable code.

There is an overwhelming number of both source languages and target languages. This leads to an even larger number of compilers. People have developed very powerful methods and programs to generate compilers and interpreters. In the MUSAC-project we used *Purdue Compiler Construction Tool Set* (PCCTS) which is a set of public-domain software tools designed to facilitate the construction of compilers and other translation systems. There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a *syntax tree* is used in which each node represents an operation and

the children of a node represent the arguments of the operation.

An *interpreter* performs the operations implied by the source program instead of producing an executable program. For an assignment statement, for example, an interpreter might build a syntax tree similar to the one depicted in Figure 7 on the facing page and then carry out the operations at the inner nodes (those with a double squared box) as it walks trough the tree. At the root $\boxed{\boxed{=}}$ it discovers it should perform an assignment to perform, so it evaluates the expression on the right side and then stores the resulting value in the location associated with the left child $\boxed{a}$. When evaluating the root's right child the interpreter discovers it should compute the product of two expressions. It would recursively compute the value of all the subexpressions. Once the assignment statement is executed the tree can be discarded. The only thing to remember is the result stored in a location for $\boxed{a}$. An interpreter does not analyze the entire input but small portions of it, which it executes subsequently.

3.2 Language M

The MUSAC-project defines its own language called M. Its purpose is to describe a particular method in the field of Analytical Chemistry. It allows to describe each and every step of an analytical measurement that contribute to the uncertainty and the result of the measurement.

The key idea pursued in M is that any object involved in measurement is defined by a list of attributes. Attributes are key-value pairs. Values can be almost anything. The MUSAC-system reads the entire description and does a complete analysis on the input. So, the MUSAC-system is neither exactly an interpreter as described in the previous section nor is it a proper compiler. It's something in between. It completes the analysis phase and keeps an internal representation (the syntax tree) of the entire input. Then different operations are executed on this syntax tree. The following sections define the syntax of M.

3.2.1 Extended Backus-Naur Form (EBNF)

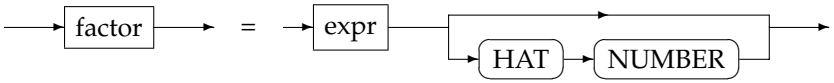
A language is defined by a grammar. Using EBNF is one way to describe a grammar. EBNF is standardized by ISO/IEC 14977 [3]. By applying *produc-*

tion rules one arrives from a sequence of *non-terminal symbols* to a sequence of *terminal symbols*. Consider for example the following rule

```
factor = expr , [ HAT , NUMBER ] ;
```

On the left hand side of "=" we have the rule name `factor` which is always a *non-terminal* symbol. On the right hand side possible replacements for the non-terminal symbol are listed. In this example the non-terminal symbol `factor` can be replaced by another non-terminal symbol named `expr` optionally followed by a terminal symbol `HAT` and a symbol `NUMBER`. Figure 8

Figure 8 Graphical Representation of an EBNF Production Rule



depicts the situation: the non-terminal symbol on the left hand side must be replaced by a valid path on the right hand side. This yields a new sequence of non-terminal and terminal symbols. All non-terminal symbols are subsequently replaced according to their production rule until we eventually end up with an arbitrarily long sequence of terminal symbols only.

The description above explains how a syntactically correct program can be generated starting from one non-terminal symbol. But the interpreter is run with a given a program, i.e. a sequence of terminal symbols and the interpreter must decide whether that sequence is syntactically correct. Prima vista this is a different task. Upon closer look it is exactly the same process: the interpreter tries to find the sequence of production rules which leads to the given program. The exact technique how this sequence can be found is not of importance here. If the interpreter fails to find a sequence then the program is syntactically wrong. If it succeeds the program is correct. Note that if there is more than one sequence leading to the given program, then the syntax itself would be ambiguous. One program statement could be interpreted in two different ways. This shall not be allowed in M. And, as a side effect, the sequence of production rules found by the interpreter yields directly the syntax tree mentioned in Section 3.1. The PCCTS tool set helps to ascertain the unambiguity of a syntax and to generate the skeleton of the interpreter.

3.2.2 Grammar of M

Table 3 shows the replacement patterns used in our variant of EBNF. The entire grammar describing M can be found in Section 3.2.5.

Table 3 EBNF Reproduction Rules' Patterns
The standardized EBNF contains several additional replacement patterns, but this subset suffices for M. For convenience brackets group items together. Furthermore the ISO requires that terminal symbols be quoted in various styles, whereas we use names in uppercase for terminal and names in lowercase for non-terminal symbols.

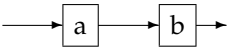
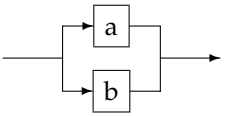
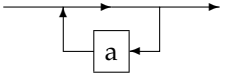
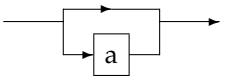
EBNF Notation	Graphical representation	Meaning
`a , b`		`a` followed by `b`
`a | b`		either `a` or `b`
`{a}`		arbitrary times `a` (incl. 0)
`[a]`		`a` or nothing (`a` is optional)
`lowercase`		non-terminal symbol (another rule)
`UPPERCASE`		terminal symbol (a token)

Table 4 Tokens in M

Token name	Token content
ASSIGN	`:=`
ASTRING	any text within double quotes (`"..."`)
COMMA	`,`
DISTRIBUTION	`:` followed by one of `(␣|g|r|t|l)`
DIV	`/`
END_UREAL	`>`
EOF	end of file
EQUAL	`=`
FUNCTION	one of the keywords `function`, `dfunction` or `sfunction`
HAT	`^`
INSTRNAME	a name starting with uppercase letter
LBRACE	{
LBRACKET	`[`
LPAREN	`(`
MINUS	`-`
MUL	`*`
NAME	a name starting with lowercase letter
NUMBER	a number with or without decimal point, with or without exponent
PERCENT	`%`
PLUS	`+`
RBRACE	}
RBRACKET	`]`
REMARK	# skips the rest of the line
RPAREN	`)`
SEMICOLON	`;`
START_UREAL	`<`
UNIT	optional prefix (`c|m|k|M|...`) followed by a SI unit (`m|g|l|mol|...`)

3.2.3 Token definitions for M

A terminal symbol, or a *token* is a sequence of characters that is read as a whole, e.g. a name, a keyword or a number etc. All tokens used in M are defined in Table 4. Observe that the description of token `UNIT` is itself already a tiny informal grammar in this table. But it cannot be handled by a proper grammar, because the "allowed" unit string are not unambiguous, e.g. `min` could be understood as prefix `m` followed by `in` for "Inch" or it could be understood as `min` for "Minute". Therefore the decision about such ambiguities is left to the lexical analyzer. It's the process reading in single characters and returning token identifiers to the grammar analyzing process. In this particular situation of reading a `UNIT` token it executes a two stage guessing. First, it tries to match the characters against a list of known units. If that fails, it tries to split off a known prefix and matches the rest against the list of known units anew (Table 6 lists known prefixes and units). If that fails too, a lexicographical error is present and the process of parsing the input failed due to an unrecognized sequence of characters. In the first two cases either a proper unit was recognized or a prefixed unit was recognized, both are returned as token `UNIT` to the grammatical analyzer.

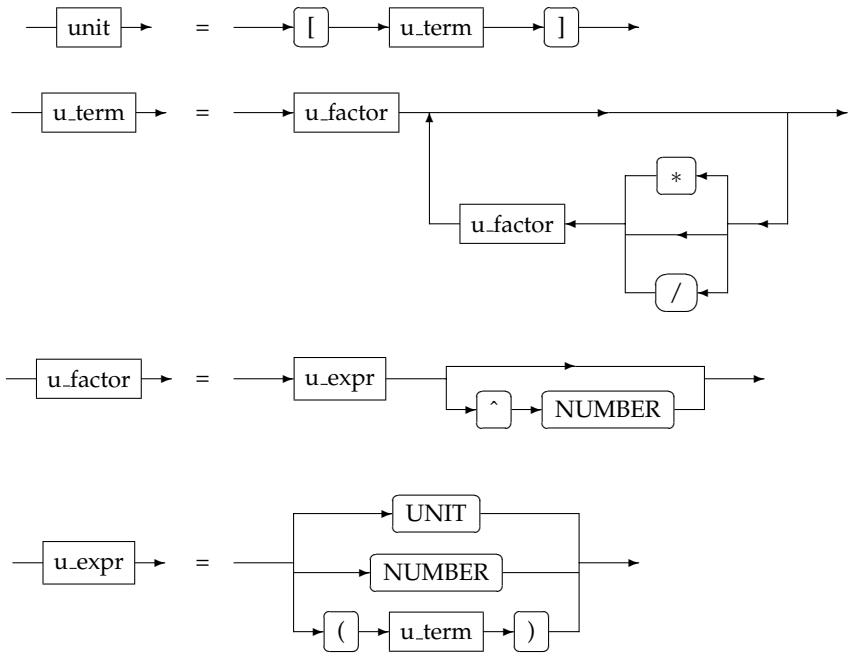
3.2.4 Example: Grammar for Units

Consider the following EBNF example describing units in M:

```
u_expr = UNIT | NUMBER | ( LPAREN , u_term , RPAREN ) ;
u_factor = u_expr , [ HAT , NUMBER ] ;
u_term = u_factor , { [ MUL ] | DIV , u_factor } ;
unit = LBRACKET , u_term , RBRACKET ;
```

Stated in words: a unit is enclosed within `[` and `]` and the expression within the brackets must only contain multiplicative terms. For example `[10*ml]` is a valid `unit`, because firstly the `[` and `]` match, secondly the `u_term` is replaced by `u_factor * u_factor`, thirdly the first `u_factor` is replaced by the terminal symbol `NUMBER`, i.e. the symbol `10`, and finally the second `u_factor` is replaced by terminal `UNIT`, i.e. `ml`. Note that `[10 ml]` is also valid since by rule `u_term` the token `*` is optional. Similarly the following units are all valid `[1/mm]`, `[mol/g mm^2]` and even `[(mol/g mm)^2]`. Note that also `[m^0.5]` is syntactically correct, but a fractional exponent for

Figure 9 Graphical Representation of the Grammar Defining a Unit in M



units makes no sense. Therefore the parser issues a warning and truncates the fractional exponent in M.

Axel Reichert states in [37]: "By the way, a very common [typographic] mistake is to place units into brackets, like $[N]$. The correct notation is $[F] = N$." In this text we will typeset units using a separate font as in 1 m in order to distinguish a bit more clearly between running text or formula and units. And we will use typewriter style for examples of M-code in order to emphasize that such text can verbatimly be passed to the MUSAC-system. The brackets around units in M are used as begin and end marker of a unit. They were introduced in M despite that common typographic mistake, because they ease the task of parsing units very much. And after all, M is meant to be read and written by programs rather than to please physicists' eyes.

3.2.5 Complete Grammar of M

This is the entire grammar of M described in EBNF as explained in Section 3.2.1.

```
start = statements , Eof ;

statements = { ( statement ,  SEMICOLON ) | SEMICOLON } ;

statement = asgmnt | eqn | term | funcdef ;

measurement = leaf , substance | NAME ,
              [ instrument | NAME , [ environment | NAME ] ] ;

ureal = START_UREAL ,
          ( [ PLUS | MINUS ] , NUMBER , [ unit ] ) | NAME ,
          Distribution ,
          ( NUMBER , [  unit | PERCENT ] ) | NAME ,
        END_UREAL ;

leaf = NUMBER | ureal , [ unit | PERCENT ] ;

named_attrlist = ASTRING | InstrName ,
                 [ LPAREN , attrlist , RPAREN ] ;

substance = named_attrlist ;
```

```
instrument = named_attrlist ;

environment = named_attrlist | ( LPAREN , leaf , RPAREN ) ;

arglist = [ arg ] , [ COMMA , arglist ] ;

arg = term | ASTRING ;

attrlist = [ attr ] , [ COMMA , attrlist ] ;

attr = eqn | funcdef ;

alist = [  attr | arg  ] , [ COMMA , alist ] ;

veclist = [ term ] , [ COMMA , veclist ] ;

vector = LPAREN , veclist , RPAREN ;

term = factor , { PLUS | MINUS , factor } ;

factor = expr , { MUL | DIV , expr } ;

expr = funccall | vector | noexpr | NAME | funcvalue |
       ( PLUS , expr ) | ( MINUS , expr ) ;

funccall = NAME | funcvalue , LPAREN , alist , RPAREN ;

noexpr = measurement | leaf | ( LPAREN , term , RPAREN ) ;

eqn = ( NAME , [ unit ] ) | funccall , EQUAL ,
      term | named_attrlist ;

asgmnt = NAME , ASSIG , term | named_attrlist ;

funcdef = FUNCTION , NAME , LPAREN , arglist , RPAREN ,
          funcbody ;

funcvalue = FUNCTION , LPAREN , arglist , RPAREN , funcbody ;

funcbody = LBRACE , statements , RBRACE ;
```

```
u_expr = UNIT | NUMBER | ( LPAREN , u_term , RPAREN ) ;

u_factor = u_expr [ CARRET , NUMBER ] ;

u_term = u_factor , { [ MUL ] | DIV , u_factor } ;

unit = LBRACKET , u_term , RBRACKET ;
```