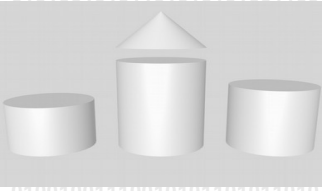

ODABA[NG]

BNF Parser Tools

run Software-Werkstatt GmbH
Weigandufer 45
12059 Berlin

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com
web: www.run-software.com

Berlin, August 2016

Content

1 Introduction

BNF parser tools provide functionality for analysing BNF based expressions. The document defines how to define problem relevant BNF but also how to change BNF syntax for special purposes. Tools are provided for generating ad-hoc or C++ BNF parser functions. In order to locate errors, several protocol and debug functions are provided.

Platforms

Parser tools are defined as platform independent C++ classes for Windows platforms as well as for UNIX platforms (Linux, Solaris).

Open Source

The Parser tools are open source and can be used for commercial as well as for non commercial reasons.

Classes

Besides parser tools, two classes are provided that allow analysing BNF based script files. Detailed function definitions are provided in class reference (ODABA Online Documentation (v.r.s) / Reference documentation / ODABA Application Program Interface / Service Classes)

`BNFParser`

The **`BNFParser`** class provides functionality for analysing script files or strings based on user-defined BNF. As result, the class provides a **`BNFNode`** tree.

`BNFNode`

The class provides functionality for accessing nodes in the tree. BNF nodes always get a type, which corresponds to the BNF symbol that classifies the data managed by the node.

2 BNF Parser

BNF classes are provided for defining expression by means of BNF, creating or generating BNF parsers based on a BNF definition and analyzing strings according to the defined syntax.

You may generate a C++ class for your parser or create an ad-hoc parser according to a given BNF. When parsing an expression the parser returns a syntax tree, that provides the values for the symbols found in the expression.

A BNF definition may refer to symbols defined in another BNF definition. This allows defining common BNF symbols e.g. for name and number (as in BNFStandardSymbols).

A string according to a given BNF syntax is based on a (top) BNF symbol. You may derive a specific BNF parsers for each type of BNF you want to support. The BNF is defined in the constructor for the BNF parser. Any number of spaces is allowed between symbols in a BNF but not required. Spaces are usually considered as separators between symbols.

Using BNF parser tools requires the following steps:

Defining a BNF Using parser classes requires a BNF definition that describes the syntax for the expressions to be parsed. Specific rules for defining a bnf syntax are described in “Defining a BNF”.

Create parser From a given BNF definition you may create an ad-hoc parser or generate a C++ class for your parser definition. Creating an ad-hoc parser is good for testing the BNF, while generating a parser class can be considered as final step.

Analysing expressions Analysing expressions for a given syntax will create a BNF data tree, which contains nodes for each symbol found in the BNF.

Processing a BNFData tree The BNFData tree contains the nodes for the symbols found in the analyzed expression. You may list the BNFData tree or use several function for extracting the data for the nodes.

Defining a BNF

Using parser classes requires a BNF definition that describes the syntax for the expressions to be parsed. The rules for defining a BNF are described by a BNF, again, which is self-describing. The BNF described here includes some practical extensions as keywords and concurrency count.

BNF Syntax

The BNF syntax (i.e. the meta-BNF) is defined as follows:

```
bnf          := bnf_stmt(*)
bnf_stmt     := definition | keyword | reference | comment_line
              | nl

definition   := sym_name def_sym rule [comment] nl
def_sym      := ':=' | '|=' | '=='
rule         := prule [ alt_prule(*) ]
alt_prule    := '|' prule
prule        := ext_symbol(*)
ext_symbol   := elm_symbol [ multiple ]
multiple     := '(' maxnum ')'
maxnum       := '*' | std_digits
elm_symbol   := sym_name | std_strings | impl_symbol |
                opt_symbol | char_set
impl_symbol  := '{' rule '}'
opt_symbol   := '[' rule ']'

char_set     := charset | ex_charset
ex_charset   := '^' charset
charset      := '(' val_list ')'
val_list     := val_def [ valdef_ext(*)]
valdef_ext   := ',' val_def
val_def      := value | val_int
val_int      := value '-' value
value        := std_digits | std_strings

keyword      := sym_name '::' keydef [ alt_keydef(*) ] nl
alt_keydef   := '|' keydef
keydef       := cstring

reference    := name '::=' symref nl
symref       := class_ref | symbol_ref
class_ref    := 'class' '(' name ')'
symbol_ref   := 'ref' '(' name ')'

sym_name     := name
name         := std_name
comment_line:= comment nl
comment      := '//' std_anychar(*)
CC           := '//'

std_symbols ::= class(BNFStandardSymbols)
std_name    ::= ref(std_name)
std_digits  ::= ref(std_digits)
std_strings ::= ref(std_strings)
std_anychar ::= ref(std_anychar)
nl          ::= ref(std_nl)
```

You may define your own BNF specification, as long as you define the symbols with red bold letters. Other red symbols are optional and can be defined in your specific BNF definition (see “User-defined BNF Syntax”).

Separators

Blanks and tabs (9) are considered as separators between BNF symbols and must not be defined explicitly. New line characters (10,13) can be defined as automatic separators as well, but here, new line characters are used as symbols and not as separators, which allows terminating a BNF statement by new line characters.

Note, each BNF statement requires a line break at the end, i.e. the last statement must be followed by a line break as well, which results in an empty line of the definition file.

comments

A character sequence introducing a comment can be defined by the CC (comment characters) symbol. Any sequence beginning with this string at the beginning of a line or after any separated symbol (symbols that can be followed by one or more separators) is considered as comment until the line end. Thus, comments can be made at each end of line or as separate comment.

```
CC := '//'
```

The CC symbol is a reserved symbol and cannot be used otherwise.

bnf

A BNF definition contains a number of BNF statements, which are symbol definitions (symdef), symbol references (symref) or comments. Each statement is terminated by line break.

definition

A symbol definition defines one or more production rules (prule) for a symbol in the BNF or an empty line. There are two define symbols supported for rule definitions.

:=

This definition symbol defines a normal rule definition. Normal production rules check keywords before looking for complex symbols. When an unexpected keyword appears, analyzing the expression terminates with error.

==

This definition symbol defines a simple symbol, which will not check keywords, i.e. keywords are accepted as valid strings and are not interpreted as keywords.

|= This definition symbol defines a break symbol. When having complex BNF definitions, errors in the string to be analysed may make analysing very inefficient, since the parser tries to evaluate all possible paths to find a solution for the problem. Break symbols may increase the performance, since an error in a break symbol causes the parser to stop immediately.

rule A definition rule may consist of any number of simple production rules separated by '|'.

alt_prule Any number of alternative production rules may follow a production rule.

prule A production rule for a symbol is a list of single or multiple symbols, which might be defined as optional.

ext_symbol An extended symbol is a symbol which may be succeeded by a multiplier.

multiple Multiple symbols is a small extension to standard BNF which allows defining a certain number or any number of symbols in a BNF definition. The following expression

```
x := symbol(*)
```

maxnum Defines any number of symbols and corresponds to

```
x := symbol [x]
```

Using multiple symbols has two advantages. One is that it becomes much easier to define specific numbers of symbols as maximum 4. The other advantage is that a multiple symbol appears as list in the BNF data tree, while the recursive definition would produce a hierarchy.

You may define multiple symbols as optional or not. Defining a multiple symbol like x(4) means, that x must appear exactly 4 times. Defining an optional multiple symbol like [x(4)] means, that x can appear maximum 4 times. Any number of symbols is indicated by '*' as number. x(*) means, that x must appear at least one time. [x(*)] means, that x may appear any number of times or not at all.

elm_symbol Elementary symbols are basic elements of the rule. An elementary symbol may refer to a defined symbol, a string constant or an implicitly defined symbol.

Impl_symbol Implicitly defined symbols are symbols, which do not get an explicit symbol name. Implicit symbols are defined as rule enclosed in { }.

```
x := symbol { 'a' | 'b' }
```

Which corresponds to:

```
x    := symbol aORb
aORb := 'a' | 'b'
```

opt_symbol — Optional symbols are restricted in this specification to exactly one symbol. This, again, does not restrict the power of the BNF but requires for complex optional expressions the definition of a separate symbol.

char_set — A character set allows defining the characters a symbol may refer to. **Character sets have to be defined before being referenced.**

ex_charset — An excluding character set defines characters not accepted for a symbol. Excluding character sets are preceded by '^'.

charset — A character set defines a list of single values or value intervals enclosed in (…).

val_list
valdef_ext — The value list is a list of values or intervals separated by comma.

val_def — A value definition is a single value or value interval

val_int — A value interval defines the lowest and the highest value. All values between lowest and highest including the limits are considered as included or excluded values (e.g. 'a'-'z' or 33-42)

value — A value is a number (sequence of digits) or a character enclosed in '..' (e.g. 'a'). Hexadecimal values are not supported

keyword — Keywords are terminal symbols, which are reserved for specific use, only. Strings defined as keywords cannot be used in other roles within a document following the BNF rules.

```
_structure :: 'structure' | 'STRUCTURE'
```

The example above defines the 'structure' keyword. Then, 'structure' or 'STRUCTURE' cannot be used e.g. as name in the document (e.g. C++ file). One may, however, use 'Structure' as name, since keywords are case sensitive.

alt_keydef — Any number of string symbols can be defined for keywords.

keydef	A keyword definition defines the keyword string. Keyword strings must not contain spaces, tabs or line breaks.
reference	Symbol references can be used to refer to external symbol definitions in other BNF definitions.
symref	A smbol reference is either a class reference (reference to other parser) or a symbol reference.
class_ref	The BNF definition that contains the symbols to be referenced, must be referred to as class reference. Referring to an external BNF definition makes all symbols defined in the external definition available in the current definition.
symbol_ref	Specific symbols in the external BNF definition can be referenced by alias names to avoid naming conflicts for symbols.
sym_name	Symbol names must start with an alphabetical character and may contain numbers, '_' and '&' in the following characters
std_strings	A string constant consists of one or more characters enclosed in '' ('+', '-', 'SELECT'). Quotes within a string constant can be defined with a preceding backslash (\'). String constants in a BNF expression acting as keywords. When being defined in a rule and not as keyword explicitly, the string constant is not reserved and can be used as e.g. name in other places.
Standard Symbols	The referenced BNF for standard symbols (BNFStandardSymbols) refers to the definition of common used BNF symbols. Standard symbols define specific character sets, numbers and name symbols as described in the subsequent BNF

```
std_symbol := std_constant | std_name | std_separator

std_constant := std_float | std_string | std_bool | std_hex
std_bool := std_false | std_true
std_false := 'false' | 'FALSE' | 'NO'
std_true := 'true' | 'TRUE' | 'YES'

std_hex := '0x' std_hexdigs
std_name := std_alpha1 [ std_nchars ]
std_compname := std_alpha1 [ std_compchars ]
std_alpha1 := std_alpha | std_nspec

std_number := std_integer | std_decimal | std_float
std_float := std_integer [std_decimalp] std_floatp
std_floatp := 'E' std_integer
std_decimal := std_integer std_decimalp
std_decimalp := '.' std_digits
std_integer := std_digits | '+' std_digits | '-' std_digits
std_line_end := [' '(*)] nl
std_stringn := '"' [std_str2(*)] '"'
std_string := '\'' [std_str1(*)] '\'' | '"' [std_str2(*)] '"'
std_str1 := std_cchar1(*)
std_str2 := std_cchar2(*)
std_cchar1 := std_dapost | std_bss | std_cchar(*)
std_dapost := \' | '' | "
std_cchar2 := std_dquote | std_bss | std_cchar(*)
std_dquote := \" | "" | '
std_bss := '\' std_bsc
std_bsc := '\' | '"' | 'n' | 't' | 'r' | 'x'

std_comment := std_combeg [std_comchar(*)] std_comend
std_combeg := 0x0101
std_comend := 0x0202
std_comchar := 1-255 except: 0x01, 0x02 (for / * and * /)
std_bnfchar := 33-255 except: ; <
std_anychars := std_anychar(*)
std_fixtext := std_ftchar(*)
std_nchars := std_nchar(*)
std_nchar := std_alpha | std_digit | std_nspec
std_compchars := std_compchar(*)
std_compchar := std_nchar | '|'
std_digits := std_digit(*)
std_hexdigs := std_hexdig(*)
std_separators:= std_separator(*)
std_separator := ' ' | std_nl | 0x09

std_digit   := 0 - 9[1]
```

[1] This and the following BNF expressions conflict with the BNF syntax and are used here to make the definitions a little bit shorter

```
std_hexdig  := 0 - 9, A -F, a - f
std_alpha   := a - z | A - Z
std_bs      := \
std_bsn     := \ std_nl
std_bsb     := \ (backslash blank)
std_anychar := 1-255 except: 0x0D, 0x0A
std_ftchar  := 1-255 except: $ \
std_nspec   := '_' | '$'
std_nl      := 0x0A | 0x0D 0x0A
std_cchar   := 1-255 except: ' " \
```

You may use alias names in you BNF to make it understandable, but you cannot use symbol names, which have already been defined in the standard BNF definition or in any other referenced BNF.

Ordering symbols

The order of symbols may play an important rule, when a symbol appears as starting symbol in several production rules. To avoid unlimited recursions and parser errors, some additional rules and suggestions for BNF definitions have been defined.

Completeness

All symbols referenced in the BNF definition (production rules) must be defined either in the BNF definition or in referenced external BNF definitions. Unresolved symbol references are shown when running the CreateParser function or when compiling the generated parser class.

Top-down

The BNF definition must be strict top-down, i.e. symbols should be defined after being referenced, or in other words: after defining a symbol it should not be referenced anymore.

An exception from this rule are character sets, which have to be defined before being referenced.

It is not necessary to follow the top-down rule in the BNF definition file, since the system will reorder the symbols later according to this rule. When the BNF contains recursive definitions like:

```
a := b
b := a
```

which cannot be resolved, the CreateParser function or the BNF parser constructor will generate an error message and the BNF should not be used before solving the problem. You may create or generate a parser for recursive BNF definitions, but it may run into problems analyzing expressions defined by such a BNF.

Priority of symbols

The BNF parser assigns a priority to each symbol in the BNF definition according to the Top-down relationship between the symbols. When there are different possibilities for resolving an expression, symbols with higher priority are resolved before symbols with lower priority.

In some cases there are different ways for setting symbol priorities. In this case the implicit priority given in the BNF definition (first symbol highest priority, last symbol lowest) is used to determine the symbol priority.

Two symbols ahead

Since BNF definitions allow using symbols as starting symbols in several production rules, ambiguity cannot be avoided.

```
x := b
y := b c
```

To make the syntax as save as possible, more specific expressions should be defined before less specific ones (i.e. y should be defined before x in this case, because x is less specific since any symbol may follow b depending on the rest of the BNF definition). The parser is using a “looking two symbols ahead” mechanism, which guarantees, that expressions can be interpreted correctly as long as ambiguous production rules differ in the second symbol.

In this case, the latest symbol will get highest priority and will be evaluated before the previously defined symbol.

Top symbol

The first line in the BNF defines the top symbol, which gives the name to the BNF. When using the parser for analysing an expression it will always start with the top symbol, unless another symbol has been defined for analysing a sub-expression. The top symbol should not be referenced at any place in the BNF. When references become necessary, another symbol should be created:

```
expression := expr_def
expr_def   := a
a          := 'a' expr_def
```

(Note, that this definition is not recursive, since expr_def is not referenced as starting symbol.)

Example

In the following sections we will consider a simple BNF for arithmetical operations.

```
operation  := operand [right_side(*)]
right_side := operator operand
operand    := number | '(' operation ')'
operator   := '+' | '-' | '*' | '/'

std_symbols  ::= class(BNFStandardSymbols)
number       ::= ref(std_integer)
```

This is a simple BNF for defining any expression with the basic arithmetical operations. The expression refers to the standard definition for integer numbers as defined in the standard symbol BNF (BNFStandardSymbols).

Optimisation

When the BNF is designed in a way that bottom-up paths are unique, i.e. between two symbols do not exist more than one path, the parsing process can be optimised. Optimising parsing may increase the parsing speed by factor 10.

```
parm       := parms option
option     := opt_n | opt_b
opt_n      := '-n'
opt_b      := '-B'
```

In the example, there are two possible ways from symbol '-' to **option**:

- → opt_n → option
- → opt_b → option

In this case, optimisation may lead to problems analysing expressions referring to both paths. The rule is not obvious and becomes necessary for optimization reasons, only. Since this is often a question of properly defining the production rules, optimisation can be requested after redefining the bnf. The example above could be resolved as follows:

```
parm       := parms option
option     := '-' opt_char
opt_char   := 'n' | 'B'
```

When the bnf definition is strict in this sense, you may call `Optimise()` after constructing the parser in order to activate optimisation.

User defined BNF syntax

There is not a real standard for writing a BNF. Nevertheless, most BNF notations are similar in principle, except the meta-symbols used in the BNF. Nevertheless, transforming a BNF into another “dialect” is a rather boring job. Hence, parser classes support alternative BNF definitions as long as the definition is based on the same symbols.

The following example shows a BNF notation used for defining the SQL-99 syntax:

```
bnf          := bnf_stmt(*)
bnf_stmt     := definition | comment_line | nl

definition   := sym_name '::=' rule nl nl
rule         := prule [ alt_prule(*) ]
alt_prule    := [nl] '|' [nl] prule
prule        := ext_symbol(*)
ext_symbol   := elm_symbol [ multiple ]
multiple     := '...'
elm_symbol   := sym_name | std_strings | impl_symbol |
                opt_symbol
opt_symbol   := '[' rule ']'
impl_symbol  := '{' rule '}'
sym_name     := '<' name '>'
name         := std_name(*)
cstring      := std_string

cstring      := std_bnfchar(*) | string

CC           := '--'

std_symbols ::= class(BNFStandardSymbols)
std_name    ::= ref(std_name)
std_string  ::= ref(std_string)
std_bnfchar ::= ref(std_bnfchar)
nl          ::= ref(std_nl)
```

For running the syntax analysis with a user defined BNF syntax, you must define a path to the file containing the BNF definition.

In the example above, symbol names may consist of several names separated by blank. One may create a parser from such a BNF as well, but one cannot generate a C++ parser class from this definition, since symbol names are used in the parser class as variable names, which must not contain blanks.

Create Parser

After a BNF definition has been provided, this can be tested in two steps. The first step is creating the parser, which will report definition errors for the BNF definition file. From a given BNF definition you may create an ad-hoc parser or generate a C++ class for your parser definition. Creating an ad-hoc parser is good for testing the BNF, while generating a parser class can be considered as final step.

Build Parser

There are two ways of building a parser, which are rather likely.

```
#include <csos4mac.h>
#include <sBNFParser.hpp>
#include <sBNFData.hpp>

int main(int argc, char* argv[])
{
   BNFParser     *bparser;
   BNFData       *bdata = NULL;
   char          *path = "arop.bnf";
   char          *accpath = "arop.exp";

   GenerateParser(path,"e:/parser.cpp");
   if ( bparser = CreateParser(path,true) )
      bdata = bparser->AnalyzeFile(accpath,true);

// list BNF data tree structure
   if ( bdata )
     bdata->Print(0,YES);

   delete bdata; // delete bdata before deleting the parser!!!
   delete bparser;

  return(0);
}
```

The CreateParser function will print a symbol priority list on the console (list option in the CreateParser function):

```
Symbol list
    'arithmetical_operation'
    'operation'
    'right_side'
    'operand'
    'operator'
    'std_symbols'
    ... other std_sybols follow
```

Since operand and right_side have the same priority, the sequence in the BNF definition determines the priority ang gives operand a higher priority since it has been defined before right_side.

Generate parser class

Calling the GenerateParser() function as in the example above will generate a C++ parser class, which can be compiled immediately. This class may replace the ad-hoc server created from the BNF definition (Create-Parser) in the example.

It is suggested creating and testing the parser before generating the parser class. The CreateParser() function allows passing a trace file name as third parameter, which will record all the attempts to resolve an expression. This is helpful for detecting errors in critical situations.

Check Expressions

In the second phase you can check several expressions with the parser created and view the result as BNFData tree. Calling the parser function Analyze as shown in the example above allows checking an expression. When successful, the function returns a BNF data tree, which can be displayed using the BNFData Print function.

Analyzing expressions

Analysing expressions for a given syntax will create a BNF data tree, which contains nodes for each symbol found in the BNF. You may analyse a complete syntax expression according to a given BNF definition but also a sub-expression.

For analysing a complete expression you may pass a filename or a 0-terminated string with the expression to the parser.

```
{
   BNFParser     *bparser = ...;
   BNFData       *bdata = NULL;

   ...

   bdata = bparser->AnalyzeFile(accpath,true);
   bdata = bparser->Analyze(string,true);

}
```

The skip option in the call (true) indicates, that the parser will skip separators at the beginning of the expression.

For analysing a sub-expression the parser must be called with the symbol name the sub-expression corresponds to. The sub-expression must be passed as 0-terminated string in this case.

```
{
   BNFParser     *bparser = ...;
   BNFData       *bdata = NULL;

   ...

   bdata = bparser->Analyze(string,"operand",true);

}
```

The symbol to be parsed is passed as symbol name. It must be a valid symbol defined for the parser or a referenced parser.

Since the parser always tries to analyse the complete expression, the string must not contain data after the end of the sub-expression. Otherwise the parser ill return an error.

Processing a BNFData tree

The result of analyzing an expression is a syntax tree, which consists of BNF data nodes. The syntax tree contains the nodes for the symbols found in the analyzed expression. You may list the BNFData nodes or use several function for extracting the data for the nodes.

Print syntax tree

The BNFData nodes in the syntax tree can be displayed using the BNFData Print function. The result for the expression

127 + 19 * (13 + 22) - (12/4)

analyzed according the sample BNF arop.bnf will return the following syntax tree (each line represents a BNFData node):

```
arithmetical_operation: 127 + 19 * (13 + 22) - (12/4) ...
  arithmetical_operation: 127 + 19 * (13 + 22) - (12/4) ...
    operation: 127 + 19 * (13 + 22) - (12/4) ...
      operand: 127 ...
        std_integer: 127 ...
      right_side: + 19 ...
        operator: + ...
          operator: + ...
        operand: 19 ...
          std_integer: 19 ...
      right_side: * (13 + 22) ...
        operator: * ...
          operator: * ...
        operand: (13 + 22) ...
          operand: ( ...
          operation: 13 + 22 ...
            operand: 13 ...
              std_integer: 13 ...
            right_side: + 22 ...
              operator: + ...
                operator: + ...
   ... and so on
```

Each node in the tree is listed with the referenced symbol name and the value for the symbol.

Evaluate a syntax tree

There are several functions provided in the BNFData class that support browsing through the syntax tree. Iterator functions provide nodes on the same level, but you may also look for a specific symbol on a certain level or recursively. More details are available in the function reference (www.run-software.com/ODABADocu)

3 Debugging BNF

Analysing syntax strings for a given bnf definition can be debugged in a limited way. Debugging BNF definitions requires a console application, which analyses the expression.

For debugging you need to set run time options, which can be set as environment variable:

```
set BNFTrace=c:/temp/trace.lst
```

or while running the application:

```
SetOption("BNFDebug","c:/temp/trace.lst")
```

BNFTrace Setting the BNFTrace option to a file path will write the analysing steps to the trace file, which may help to detect syntax loopsor inefficient definitions.

BNFDebug Setting the BNFDebug option to YES will display the parsing steps on the console. There are a few commands allowing to control the debugging process.

Enter	Analyse the next BNF item.
number	You may enter a number to skip the next 'number' of parsing steps. Entering a number will reset error or break commands.
e[rror]	The debugger stops, when an error has encountered.
b[reak]	The debugger stops at the next break symbol.

Output The debugger displays the analysed BNF symbols and the symbol names before the prompt character (>). The symbols names are indented according to their position in the symbol hierarchy.

Error	When an error has been encountered for the symbol, an asterisk (*) is displayed in front of the symbol name.
Break symbols	Break symbols are marked by a plus (+) in front of the symbol name.

4 Example

An example for arithmetical expressions as being defined above could look as follows. In order to evaluate the expression, we provide a hierachical operation tree:

```
struct     Operation {
   public: int32      value;   // operand value
   public: Operation *oper;    // Left operand
   public: Operation *next;    // Right operand
   public: char       op;      // Arithmetical operation
};
```

In order to analyse and evaluate the expression, a class ArOperation has been provided, which creates the parser, analyses the BNF tree and evaluates arithmetical expressions entered via command line:

```
int main (int argc, char *argv[] )
{
  ArOperations    arOp;
  char            expression[1000];
  int             rc = 0;
  while ( gets(expression) ) {
    if ( !expression[0] )
      break;
    if ( arOp.Analyze(expression) )
      printf("%d = %s\n",arOp.Execute(),expression);
  }
  return(rc);
}
```

Create parser In order to create the parser we use the BNF definition as embedded character string. The parser will be created in the `ArOperations` constructor:

```
     ArOperations :: ArOperations ( )
     : oper(NULL), parser(), node()
{
#include <samples/h/arop.bnf>
// arop.bnf
// operation  := operand [right_side(*)]
// right_side := operator operand
// operand    := number | '(' operation ')'
// operator   := '+' | '-' | '*' | '/'
//
// std_symbols  ::= class(BNFStandardSymbols)
// number       ::= ref(std_integer)
  parser.create(arop_bnf);
}
```

Create BNF tree In order to create the BNF tree, `ArOperations::Ana-`

lyze is called:

```
bool ArOperations :: Analyze (const odaba::String &sExpression )
{
  bool    bState = true;
  Delete(oper); // remove last operation
  oper = 0;

  try {
    node = parser.analyzeString(sExpression);
    node.print("arop.tree",true); // print bnf tree
    oper = AnalyzeOperation(node.toSymbol("operation"));
    node.release();
  } catch ( odaba::Exception e ) {
    printf("Parser error - could not analyze expression\n");
    printf("last error: %s\n",parser.lastError().area());
    bState = false;
  }
  return(bState);
}
```

The parser function analyzeString() creates a BNF tree. In case of errors, the function throws an odaba exception. When the BNF tree has been creates, the top node is returned in node.

Create operation hierarchy

From the BNF tree the operation hierarchy is created by analysing operations and operands:

```
Operation *ArOperations::AnalyzeOperation(const BNFNode &bNode )
{
  Operation    *arop = Create();
  Operation    *last;
  BNFNode       right_side;
  int32         rightCount = bNode.count()-1;
  int32         indx0 = 0;
  if ( bNode.symbol() == "operand" )
    AnalyzeOperand(arop,bNode);
  else {               // operation
    AnalyzeOperand(arop,bNode.get("operand"));
    while ( indx0 < rightCount ) {
      right_side = bNode.get(++indx0);
      last = arop;
      arop = Create();
      arop->oper = last;
      arop->op = right_side.get("operator").value().index(0);
      arop->next = Create();
      AnalyzeOperand(arop->next,right_side.get("operand"));
    }
  }
  return(arop);
}
```

```
bool ArOperations::AnalyzeOperand (Operation *arOperation,
                                   const BNFNode &bNode )
{
  BNFNode    bnode = bNode;
  Operation *op;
  bool       bState = true;

  if ( bnode.isSymbol("std_integer") ) // number)
    arOperation->value = bnode.value().toInteger();
  else {
    if ( !bnode.isSymbol("operation") )
      bnode = bnode.get("operation");
    arOperation->oper = AnalyzeOperation(bnode);
  }
  return(bState);
}
```

The `Create()` function is called in order to create and initialize new operation elements. Since the expression syntax has been checked, in this phase errors resulting from invalid expressions can be excluded. BNF nodes always appear in a sequence according to the BNF definition and BNF elements (symbols) can be accessed via symbol names.

Evaluate expression

After the operation hierarchy has been setup, the expression can be evaluated::

```
int32 ArOperations :: Execute ( )
{
  int32    iValue = Value(oper);
  return(iValue);
}

int32 ArOperations :: Value (ArOperations::Operation *arOpera-
tion )
{
  int32    iValue = 0;
  int32    iLeft  = 0;
  int32    iRight = 0;

  if ( arOperation ) {
    iLeft  = Value(arOperation->oper);
    iRight = Value(arOperation->next);
    if ( arOperation->oper && !arOperation->next )
      iValue = iLeft;
    else if ( !arOperation->oper && arOperation->next )
      iValue = iRight;
    else switch ( arOperation->op ) {
      case '+' : iValue = iLeft + iRight;
                 break;
      case '-' : iValue = iLeft - iRight;
```

```
                  break;
     case '*' : iValue = iLeft * iRight;
                  break;
     case '/' : if ( right == 0 )
                  printf("Division by 0");
                else
                  iValue = iLeft / iRight;
                break;
      default : iValue = arOperation->value;
    }
  }
  return(iValue);
```

The result will be returned to the main function for printing..

5 GenerateParser Utility

The GenerateParser utility allows creating a parser class for a specific BNF. This may reduce the analysing time since the parser need not to be built at runtime.

GenerateParser

The GenerateParser utility can be called with the following parameters:

GenerateParser *def_path*
[*cpp_path* [*trace_path* [*bnf_path*]]]

def_path

The definition path point to the location, where the BNF-file for the parser to be generated is stored.

cpp_path

The path refers to the location, where the generated C++-file will be stored. When the file does already exist, it will be replaced. When no cpp_path or NULL has been defined, the generated C++-file is stored at the same location as the def_file replacing the def_file extension with .cpp.

trace_path

When defining a trace path, the parsing steps of the process are recorded. This allows checking, what the parser has tried to analyse the def_file, which is important in some cases, when an error occurs. When no trace path or NULL is defined, no protocol is created.

bnf_path

A bnf_path can be passed to the function, when the BNF specification does not follow the standard definition of this BNF parser. This allows analysing imported BNF syntax, which may follow different rules. In this case, a BNF-definition as described in “User defined BNF syntax” must be provides at the location the bnf_path is pointing to.