EBNF+

mapping input to output

rev	*date*	*author*	*remark*
0V1	240710	HW	a first draft based onf EBNF+V0.13 x64

Table of Contents

introduction

EBNF+ is a computer language specifically suited for mapping input to output by a combination of syntax and semantic rules. The syntax rules are in Extended BNF[1] embedded with semantic expressions. At execution input is checked against the syntax rules which control the semantic expressions to generate output.

For example consider an input of 5 characters "Hello". Such string conforms to the syntax:

```
rule = "Hello";
```

If we want to respond to this specific data we can extend above rule with some actions::

```
rule = "Hello", pOut(“Hey! How are you?”);
```

which simply says if `"Hello"` is encountered then the EBNF+ function `pOut` will output `“Hey! How are you?”`. In case something else is encountered nothing is generated and the rule is returned false to the caller. The used comma in this rule is the concatenation syntax element (read *and*) that indicates to continue the evaluation of the rule although it is here followed by a function. In other words, EBNF+ just extends the Extended BNF with semantic expressions. Hence, the plus sign in its name.

If we like to respond to an alternative content, say `"Hallo"` (Dutch for “Hello”) then a syntax rule to cover the alternative is::

```
rule = "Hello"| "Hallo";
```

where the bar (|”) is read as *or*.

Mapping the alternatives to separate output can be expressed as:

```
rule = "Hello", pOut(“Hey! How are you?”)|
       “Hallo", pOut(“He! Hoe gaat het met je?”;
```

Exception handling becomes obvious:

```
rule = "Hello", pOut(“Hey hello! How are you?”)|
       "Hallo", pOut(“He! Hoe gaat het met je?”|
       {b},     pOut(“Scusi, non ho capito!”;
```

Here {b}, flushes all input and the system will generate the remark it did not understand the input.

EBNF+ supports the standard syntax elements, while the semantic can be expressed by EBNF+ functions including the host language , that is the language to which EBNF+ sources are cross compiled to (e.g. x64 assembly).

The EBNF+ offers solutions for a large class of mapping problems from simple to complex. Typical (and proven) applications include translators (such as between Infix and Postfix expressions, hex tables and binary formats, binary payment protocols to man readable formats), filters (e.g. for finding telephone numbers, email addresses, card iso numbers), converters (e.g. between data sets in CSV, XML, Json and Yaml), interfaces (e.g. Web/API servers and clients), and compilers (that is itself! EBNF+ to assembly).

---

1 ISO14977 Extended BNF is a concise and exact way to define grammars of simple expressions or more complex such as protocols and computer languages. https://en.wikipedia.org/wiki/Extended_Backus–Naur_form

syntax

The syntax symbols in EBNF+ follow ISO14977.

Notation for syntax expressions			
symbol	***used for***	***read as***	***Example***
=	syntax rule	is defined	answer = yes | no;
{ }	repetition zero or more	zero or more	`string = {character};`
{ }*	repetition one or more	one or more	`integer = {digit}*;`
[ ]	option	with option	`yes = 'Y', ['es'] | 'y', ['es'];`
( )	grouping	first do	`yes = ('Y'|'y'), ['es'];`
|	alternative	or	`vowel = 'a'|'e'|'i'|'u';`
-	exception	except	`consonant = letter – vowel;`
+	concordance	also	`digit = character + number;`
,	concatenation	and	`integer = digit, {digit};`
' '	terminal	string	`doubleQuote = '"';`
" "	terminal	string	`lastQuote = "'Help!'";`
*	factoring	times	`tripleA = 3*"A";`
^	terminal as single byte	ascii	`carriageReturn = ^13;`
0x	terminal as single byte	hex	`carriageReturn = 0x0d;`
;	end of rule	end	`emptyStatement = ;`

..	ranges	to	`letter = 'A'..'Z' | 'a'..'z';`
			`digit = '0'..'9';`
			`character = ^0..^255;`

#	state machine rule	is defined state machine as	`payment # msg1200, msg1210;`
/* */	comment	comment	`/* This is a comment of 36 characters */`

For example an EBNF+ application to check if an *url* is well formed can be programmed in only a few clear lines:

```
url  = ["http://", host], "/", path;
path = {tChar| "/"}*;
tChar= a|n|"!"|"#"|"$"|"%"|"&"|"'"|"*"|"+"|"-"|"."|"^"| "_"|0x60|
       "|"| 0x7E;
a = "A.."Z"| "a.."z";
n = "0.."9";
```

Above *url* definition is copied from RFC7230. Unabridged it can be used in the app for parsing url's. Here, definition hence becomes implementation. Actually EBNF is often available for data structures, protocols and computer languages. Just copy them into the EBNF+ application and it will be parsedto the detail..

Also the EBNF+ syntax definition is a copy from the Extended BNF syntax as proposed by the ISO/IEC 14977 . EBNF+ proofed that they did a good job :).

singularities

EBNF+ has some interesting properties:

- Traditionally BNF grammars often are explained in their own grammar. In EBNF+ not only its own grammar defines itself but also its semantics. This makes both definition and operation of EBNF+ quite transparent.
- EBNF+ semantics is covered by a few (<100) small kernel routines. Porting, changes to syntax and code optimization are pretty straightforward.
- Despite that the compiler optimizes its generated code, source is under 50kB.
- The "|" (*or*) in EBNF+ has not the communicative property as opposed to formal BNF. For example the input "ab" is conform the EBNF+ rule "ab"|"a". But the same input will be denied if the rule would be "a"|"ab". Rules in EBNF+ are always applied from left to right in EBNF. When an operand in an *or*-expression is parsed OK, then its further operands will not be evaluated. This may require some attention by the programmer but simplifies implementation and above all avoids the ambiguities as with the traditional forms.
- In EBNF+ the input is not tokenized as traditionally done. Instead compilation is done in a single parse.
- Parsing in EBNF is byte oriented. The word-based space-delimited approach of traditional BNF is not implemented. Input is simply treated as strings of zero or more bytes allowing binary input and program languages where text format has meaning (such as in occam, yaml or expressions that close at end of line such as remarks).
- Non-context free grammars are often feasible to implement, since host language code can tweek the parser context parameters.

semantics

Actions can be programmed using the EBNF+ kernel functions but also mixed with the host language, x64. To embed the code in the EBNF syntax rules, the code must be wrapped with the `'<.'` and `'.>'` - symbols. The code including these symbols is named the code-clause.

An example is the assembly file that must be includd for every EBNF+ application:

```
<.%include "../EBNF/EBNFix.asm".>
```

This file contains all assembler directives for linking the application to the EBNF kernel functions.

The syntax of the code-clause is:

```
pCode = {'<.' , {^0..^255-'.>'},'.>' , {^09| ^10| ^13|' '}}*;
```

The definition is enclosed by *repetition one or more, using the* `{ … }*` construction. This defines that the pCode can be one or more <. … .> consecutive constructions optionally separated by white spaces and/or tabs and returns as indicated in by the `{^09| ^10| ^13|' '}-syntax part.`

The syntax part: `'<.' , {^0..^255-'.>'},'.>'` is a typical construction form that can be used for strings that are within specific signs, In the EBNF+ source these strings beside code-clauses, are comments and terminals.

The semantic part of the code-clause rule can be specified by the host language embedded in the code clause syntax:

```
pCode = {'<.' , {^0..^255-'.>'},          <.     call pOut.>
                                          <.      db 1, 10.>
                                          <.     call pOutLI.>,
                                    '.>' , {pCtrlChar|" "}}*;
```

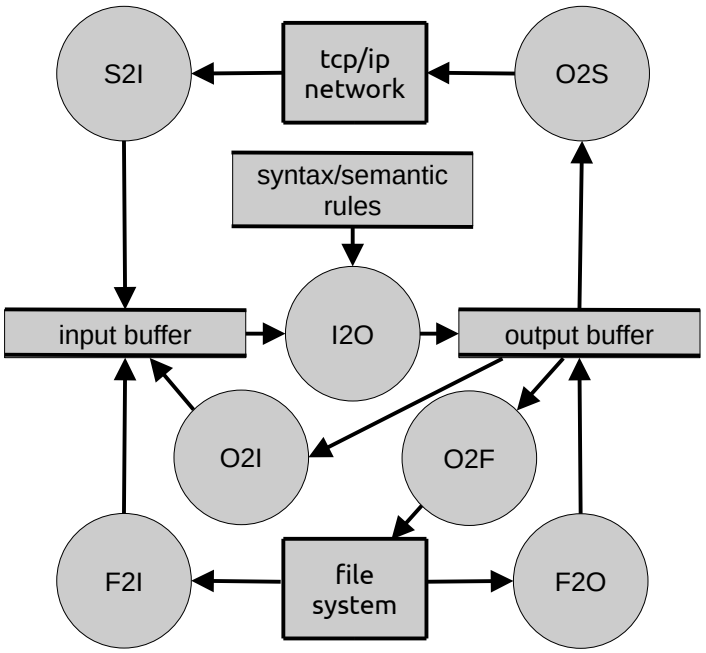
Here, the pOut routine will place a return in the output. While the pOutLI routine (parser output Last Input) will append the last parsed string to the output. The Last Input string is defined by the {^0..^255-'.>'}-syntax part. So all text between the '<.' and '.>' will be appended to the output buffer, which is the purpose of the code-clause.

IO handling

EBNF+ can handle files and TCP/IP messages in 3 consecutive steps:

a) use *F2I and the S2I*-function to transfer resp. files and messages into the input buffer
b) map by the rules this stored data into the output buffer using the *I2O*-function
c) store the data to a file or send as message using the *O2F* or *O2S*-function.

For juggling around transfers e.g. for multiple parses or using template files, one can use also *F2O* and *O2I*-function. Below the data flow of the available transfer types (circles are processes, half bold boxes are stores, full box boxes are external systems):



File handling is most straight forward. For example an EBNF+ app that inputs the file “input.txt” and maps it to an output file “output.txt” under some rule:

```
Rum =  F2I(“input.txt”), I2O(*rule), O2F(“output.txt”);
rule = "Hello", pOut(“Hey hello! How are you?”);
```

where F2I stands for “File to Input”, I2O “Input to Output” as by rule, and O2F stands for “Output to File”. Rum stands for “Run me”, a reserved EBNF+word to start execution of an application . The above 2 lines is a complete EBNF+ source. It can be compiled and executed.

TCP/IP messaging needs some initialisation of the ports. Below a primitive web server using concurrent workers to respond to GET requests.

The main control part:

```
<<Port80 resb 16>>
myserver= ServerSockOpen (*Port80, 80),
             4*Thread(*worker), WaitSecond(60),
          ServerSockClose(*Port80);
```

Here *myserver* is specified in the format of a BNF rule, but morphed into a routine specifying actions. Here, the “*” in “4*” is a repetition syntax symbol but used in semantic sense to start 4 worker threads. The “*” in the operands”*Port80” and “*worker” in the functions indicate to take the address of the operands. Port80 represents a 16 byte value declared in the host language x64 (in NASM format).

The 60s wait time is to allow the threads to do their job.

The worker source that handles the GET requests:

```
worker  = { S2I(*Port80),
            ( I2O(*htmlGET)| F2O("HttpRsp/404.html") ),
            O2S(*Port80) };
htmlGET = "GET", ws,
          ( url,        F2O(sConcat("./HttpRsp",sLI)) |
            "/ ",       F2O("./HttpRsp/home.html")    |
                        F2O("./HttpRsp/404.html")     );
```

The worker checks if the input is a GET request in which case it outputs (by F2O, File to Output) the resource. sLI stands for “string Last In”. It returns the last evaluated string in the syntax rule, here the url. sConcat will append this url to the string “./HttpRSP”, which is the programmer chosen name of the directory that contains the html resources.

Note that the curly brackets in the worker rule are syntax symbols, meaning that the enclosed expression will repeat zero to many times as long as the expression delivers a positive parse, which is here expected.

speed and code size

EBNF+ sources are compiled to a host language. Executable (Version 0.13) is around 70kB. The x86 implementation is pretty fast. Size (Version 0.13) is around 70kB. No benchmarks yet done.