Executable Grammars

Peter M. Cashin

Abstract

A BNF style grammar is a great way to define a syntax, but the BNF rules need to be translated into a parser that is implemented in some particular programming language. It has been rare to find grammar rules that are directly executed as a parser.

Contrast this with regular expressions that programmers use in their every-day programs. The translation into a parser is fully automated, regular expressions are directly executed.

This note discusses the design of BNF grammar rules that can be directly executed, gives examples, and discusses some of the implementation issues.

Introduction

BNF style grammar rules have a long history and come in many variants, for example ABNF [1], and EBNF [2]. Traditional grammar rules define how to generate all valid strings. But grammar rules can also be interpreted as analytic pattern matching rules, as in a PEG, Parser Expression Grammar [3].

For direct execution of BNF rules it is easier to interpret the rules as pattern matching rules, where the rules have a procedural interpretation as well as a declarative interpretation. The BNF rules can then be thought of as a special purpose programming language.

I'll call a BNF that can be directly executed a PBNF, for Parser-BNF. It is desirable for a PBNF to be as close to a traditional BNF as possible. But it must be amenable to a fully automated parser. To be practical the direct execution must provide a useful alternative to a custom parser, and preferably it should come close to matching the performance of a custom parser for simple every-day grammars.

The execution of a grammar must resolve any ambiguity. One way to do this would be to generate all possible syntax trees, and then select one: maybe the first one, or the largest one.

Rather than generate all syntax trees before picking one we can resolve the ambiguity at a rule level. Each rule can be specified to match an input string in only one unique

way (or fail). We can go further than this and resolve ambiguity for each operator in a grammar expression. This is how a PEG works.

Grammar operators that resolves ambiguity are not restricted to the PEG operators. A PBNF can provide a longest choice operator in place of (or in addition to) the PEG first priority choice operator. It can also provide an operator to define a context sensitive grammar, so we are not restricted to a traditional CFG (Context Free Grammar) [4].

Fortunately the majority of every-day grammars do not require anything exotic. For example, the IETF has successfully used the simple ABNF grammar language for many years. There are dozens of ABNF specifications, and none of them require complex grammar rules. They can be quite easily translated into a PEG which can be implemented with an fast top down recursive descent parser. So a version of a PBNF grammar that provides the PEG grammar operators is a good starting point.

For direct execution the grammar rules must also specify the syntax tree that a parser should generate as output. Given this a universal parser can automatically parse any language that the PBNF grammar rules can express.

PBNF Language Power

Fortunately we do not require very many operators to cover most practical grammars. A simplified interpretation of the standard operators is sufficient:

```
x | y   -- the longest match of x or y.
x*      -- the longest sequence of x that can be matched.
!x      -- not x, fails if x can be matched.
```

The other standard operators can be defined in terms of these operators:

```
x+    => x x*
x?    => x | ''      -- '' is an empty match
x / y => x | !x y    -- the PEG priority choice
```

This does not mean we should not implement these operations directly for an efficient parser. It just means they don't add any extra power for the grammar to express other languages.

The expressive power with these operators is greater than a PEG, and in practice it also covers CFG grammars. However, there are a few practical grammars that can not be expressed as a CFG (or a PEG), but a PBNF can be extended to cover them.

To allow some useful context sensitive grammars we can add operators that look back into the (partial) syntax tree generated from the preceding input string:

```
@x    -- true if x has been matched as a prior sibling/parent.
@=x   -- match the same string that the prior x matched.
```

Clearly the fewer operators the better, and it seems that this small set covers most practical needs.

Given that each operator has a single match then a PBNF grammar can be implemented with a simple top down recursive descent parser. But the direct execution of a grammar also requires a definition of the syntax tree that the parser should generate. This is discussed next.

Implicit Syntax Tree Definition

The implicit specification of a syntax tree is quite intuitive. For example, here is a grammar for the sum of a list of natural numbers:

```
sum   = num ('+' num)*
num   = digit+
digit = '1'..'9'
```

The syntax tree for "1+23+4" is a flat list:

```
sum( num(digit(1)), num(digit(2),digit(3)), num(digit(4)))
```

The syntax tree nodes have a tag corresponding to the rule name that generated it, and rules that are matched as a component part of a host rule appear as children nodes. Leaf node rules, such as digit, generate a node that tags a substring of the input character string. Of course we can implement this syntax tree structure in different ways in different programming languages, but the basic tree structure is well defined.

Clearly different grammar specifications of the same input language will generate different trees. We can also include rule notations to prune the tree structure, and enable a faster parser. In particular a terminal or leaf rule can be defined with a colon in place of an equals. The language defined by the grammar remains identical, only the output syntax tree structure is changed:

```
sum   = num '+' sum | num
num   : digit+
digit : '1'..'9'
```

The syntax tree for "1+23+4" is right associative (1+(23+4)).

```
sum(num(1), sum(num(23), num(4)))
```

Notice that the '+' is a literal match that does not appear in the syntax tree (it could do if we gave it a rule name), and the num is now a terminal string matching leaf node which uses the digit rule internally.

A PBNF may allow left recursion:

```
sum   = sum '+' num | num
num   : digit+
digit : '1'..'9'
```

The syntax tree for "1+23+4" is left associative:

```
sum(sum(sum(num(1)),num(23)),num 4)
```

Thus a sum can be defined with three different implicit syntax tree structures: a flat list (from a repeat operator), and a right or left recursive tree. A flat list is often a good choice, and an application can then process it with a right or left association as appropriate.

An Example

A practical example of a grammar that has been implemented in many different languages is the Json [5] data language specification. Here is a PBNF version of the published Json specification:

```
json   = s (object / array) s
object = '{' s pairs? s '}'
pairs  = pair (s ',' s pair)*
pair   = string s ':' s val
array  = '[' s vals? s ']'
vals   = val (s ',' s val)*
val    = object/array/string/number/true/false/null
string = quot_ (chs/esc)* quot_
esc    = bs (code / 'u' hex hex hex hex)
number = neg? digits frac? exp?
frac   : '.' int
exp    : ('e'/'E') sign? int
digits : '0'/'1'..'9' digit*
int    : digit+
digit  : '0'..'9'
sign   : '+'/'-'
neg    : '-'
true   : 'true'
false  : 'false'
null   : 'null'
hex    : digit/'a'..'f'/'A'..'F'
code   : bs/fs/quot/'b'/'f'/'n'/'r'/'t'
bs     : 92   -- back-slash
fs     : 47   -- forward-slash
```

```
quot   : 34   -- quote
s      : (9/10/13/32)* -- white space
chs    : char+
char   : 0x20..10ffff^quot^bs
-- any Unicode except control codes, quote or back-slash
```

It would hardly be noticed if this PBNF version was used in place of the original specification, but this one can be executed as a parser to generate a syntax tree. An application can easily transform the syntax tree into native language elements.

In practice a PBNF can be used to replace any ABNF or EBNF grammar that defines an unambiguous computer language.

A Practical Implementation

The PBNF rules can be interpreted directly, but for better performance we can use an abstract machine designed to execute the grammar rule operators. A compiler translates the grammar rules into parser machine instructions.

Implementation of a parser machine is simplified when the PBNF operators have a single match. The only tricky issues are the extensions of the core machine to support a memo cache for rule results (to make a packrat parser), and the extensions needed to support left recursion. The problem with using a memo cache is to avoid imposing any significant overhead on the many simple grammars that can not take any advantage of memos. The problem with left recursion is the ability to handle indirect and nested left recursions, it is a significant extra complication.

The bulk of the code to implement a PBNF interpreter is the compiler that takes the PBNF grammar rules and generates instructions for the parser machine. The parser machine instructions are designed to implement the PBNF grammar language operators (sequence, choice, repeat, literal, etc), so the code generation itself is quite straightforward.

Many grammar rules boil down to the definition of a set of characters to be matched. For example:

```
hex : digit | 'A'..'F' | 'a'..'f'
```

Because this is a terminal rule the compiler can combine the digits and letters into a single character set, as a list of integer ranges, which can be matched with a single parse machine instruction.

GIST (Grammar Into Syntax Tree) is an implementation of a PBNF parser [6]. There have been many iterations of Gist systems implemented in several different programming languages.

The parser machine in Gist includes memos and left recursion, yet it is only about 400 lines of Java code. It is quite simple to re-implement in other programming languages. It is interesting to note that because the parser machine uses simple integer arrays the Java version of the parser machine has very similar performance to the C version.

The parser machine to bootstrap Gist can be implemented with just 8 instructions (/, *, +, ?, etc), and a full Gist parser requires less than 20 instructions. A full implementation of Gist in Java with a reasonable compiler takes under 2000 lines of code.

Summary

We can directly execute grammar rules without compromising the succinct expressive power of traditional BNF style grammar rules. This allows clean neat grammar specifications that can be used directly in software applications regardless of the programming language.

It is possible to design a PBNF grammar language with an expressive power that is better than a CFG, but also enables a simple automated top down recursive descent parser.

The interpretation of the grammar rules must be extended to provide an implicit definition of the syntax tree that should be generated. Fortunately this requires very little extra notation to be added to the familiar BNF grammar notations.

An application programmer should be able to use grammar rules just as easily as regular expressions.

References

[1] ABNF http://en.wikipedia.org/wiki/Augmented_Backus-Naur_Form
[2] EBNF http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form
[3] PEG http://en.wikipedia.org/wiki/Parsing_expression_grammar
[4] CFG http://en.wikipedia.org/wiki/Context-free_grammar
[5] Json http://www.json.org
[6] Gist http://github.com/spinachtree/gist