Lingukit

Jan Bernitt

May 11, 2014

Abstract

Lingukit is a concrete formal grammar to describe the syntax of another target grammar. It is close to the model of *Grammars as Data and Solvers* and has concise declarative syntax similar to `BNF`.

1 Quick Start Guide

1.1 Matching Bytes

The grammar is designed to be used on `UTF-8` encoded input. Therefore the basic unit of a character is a `byte`.

The simplest way to match one or more bytes is to literally give a fixed sequence of characters to match. This is done by enclosing them in single ticks `'` like `'a'` for literally a or `'keyword'` for literally keyword. No escaping is supported or necessary - a single tick `'` itself is given similarly `'''`.

```
literally-a = 'a'
literally-keyword = 'keyword'
```

Non ASCII characters or white-space should instead be given as a uni-code literal in the form `\uXXXX`

To match subsets of characters a few different constructs can be used and combined.

```
digit = '0'-'9' % range
```

1.2 Matching Structure

A rule's patter or structure can be described using the common generic building blocks of information processing:

- iteration
- sequence
- selection
- (plain blocks)

TODO completion

1.3 Capturing Matches

By splitting a grammar into named rules also the blocks of the resulting parse-tree are described.

```
foo = bar baz
```

The above rule will capture a tree structure like this

```
foo
    bar
    baz
```

Also individual elements in a rule declaration can be named inline:

```
range = number:low '-' number:high
```

The above rule aliases the rule number to low and high. The resulting parse-tree will look like

```
range
    low
    high
```

Lastly rules can be referenced without also capturing them. This is used to reuse patterns that themselves do not describe a complete interesting value.

```
number = -digit+
```

The above rule says that a number consists of 1 or more digits but by using minus `-` prefix on the rule's name so this block isn't captured itself. The resulting parse-tree will just contain one token number. As a convention also all rules having a name starting with a `\` will not be captured. This is e.g. utilised to declare a rule named `\n` that can be used as an alias to the `\u000A` without capturing such a rule.

2 Grammar Specification

2.1 Names and References

A name is any sequence of ASCII letters (both lower and upper), ASCII digits, underscore `_` and dash `-`. In addition a name may start with backslash `\`. Names starting with dash `-` or backslash `\` are never captured. When referencing a rule the reference name may use a starting dash to not capture even thou the referenced rule does not start with a dash.

2.2 Short-hands

There is some *syntactic sugar* that does not add more expressiveness but better readability by giving frequently used patterns a short-hand syntax.

Occurrence

- `+ = x1+`
- `* = x0+`
- `? = x0-1`
- `[XYZ] = (XYZ)? = (XYZ)x0-1`

Sets of Characters

- `9 = '0'-'9'`
- `7 = '0'-'7'`
- `1 = '0'-'1'`
- `# = { 9 'A'-'F'} = { '0'-'9' 'A'-'F'}`
- `@ = { 'a'-'z' 'A'-'Z' }`
- `$ = { \u0000-\u7FFFFFFF }` (that is any UTF-8)

White-space

- `_ = \s = { \t \n \r ' '}`
- `, = _* = { \t \n \r ' '}*`
- `~ = _+ = { \t \n \r ' '}+`
- `. = >>* { \n \r }+ >>*`[1]
- `>> = { \t ' ' }`

[1] this is not fully equivalent as the pattern allows that CR/LF do not occur if the end of the file is reached

Listing 1: Lingukit given in its own syntax

```
grammar       : member (, member)*
member        : comment | rule
comment       : '%' (!\n+):text
rule          : name, ('=' | (':' ':'? '='?)), selection ';'? .
selection     : sequence (, '|' >> sequence )*
sequence      : element ( >> element )*
element       : (distinction | completion | group | option | string |
    terminal | ref ) occurrence?
distinction   : '<'
completion    : '..' capture
group         : '(', selection, ')' capture
option        : '[', selection, ']' capture

occurrence    : 'x'? num:min {'-' '+'}:to? num:max? | qmark | star |
    plus
num           : 9+
qmark         : '?'
star          : '*'
plus          : '+'

ref           : name capture
name          : '-'? '\'? {'A'-'Y' 'a'-'y'} {@ 9 '_' '-'}*
capture       : [':' name:alias ]

string        : ''' !'''x2+ '''

terminal      : pattern | ranges | figures
pattern       : not? (gap | pad | indent | separator | wrap)
figures       : '{', -figure (, -figure )* '}' capture
figure        : ranges | name

wildcard      : '$'
symbol        : ''' $ '''
code-point    : 'U+' #x4-8
literal       : code-point | symbol
range         : literal, '-', literal
category      : 'U+{' @+ '}'
ranges        : not? (wildcard | letter | upper | lower | digit | hex |
     octal | binary | category | range | literal | whitespace |
    shortname )

letter        : '@'
upper         : 'Z'
lower         : 'z'
digit         : '9'
hex           : '#'
octal         : '7'
binary        : '1'
not           : '!'
whitespace    : '_'
gap           : ','
pad           : '~'
wrap          : '.'
indent        : '>>'
separator     : '^'

shortname     : tab | lf | cr
tab           : '\t'
lf            : '\n'
cr            : '\r'
```

3 File Format Specification