Formal Languages as Data and Solvers

Jan Bernitt

May 10, 2014

Abstract

This paper describes the semantics of a programming language independent system to model a formal language as data in combination with the formal process of a generic solver controlled by a grammar to parse documents formulated in the grammars syntax into a index overlay parse-tree.

1 Formal Grammar as Data

A formal grammar is modelled using the same universal data structure for all target languages. This data is *interpreted* by universal solvers (e.g. a parser). No behaviour is attached to the grammar nor does it require to generate code for the production rules or the parse-tree.

1.1 Composition of Rules

A grammar is a set of production rules. Each production rule is composed out of rule components. There is a fixed set of different kinds or types of rules.

```
type Grammar = [Rule]
data Rule
        = Literal UTF8String
        | Terminal [CodePointRange]
        | Pattern UTF8String -> Position -> Length
        | Sequence [Rule]
        | Selection [Rule]
        | Iteration { r :: Rule, min :: Count, max :: Count }
        | Completion { subsequent :: Rule }
        | Capture Name Rule
        | Reference Name
data CodePointRange
        = Character CodePoint
        | NotCharacter CodePoint
        | Range { min :: CodePoint , max :: CodePoint }
        | NotRange { min :: CodePoint , max :: CodePoint }
type CodePoint = Word32
type UTF8String = [Word8]
type Position = Int32
type Length = Int32
type Name = String
type Count = Int
```

The three terminal rules `Literal`, `Terminal` and `Pattern` match bytes or codepoints of `UTF-8`. The non-terminal rules `Sequence`, `Selection`, `Iteration`, and `Completion` describe the nesting or structure. A `Capture` decoration rule is used to name a rule for reference and as a element having that name in the resulting parse-tree. The `Reference` finally allows the reuse of named rule components.

1.2 Types of Rules

Matching Bytes

`Literal`	Matches an exact sequence of UTF-8 bytes.
`Terminal`	Matches ranges of UTF-8 code-points.
`Pattern`	Matches an abstract pattern of UTF-8 bytes. The length of matching bytes is given through a particular algorithm for a particular pattern. This is the only non-concrete building block.

Matching Structure

`Sequence`	Wraps two or more components that have to sequentially follow each other. Matches if all its components match.
`Selection`	Wraps two or more alternative components. The alternatives are ordered from highest to lowest priority. Matches as soon as highest yet tried component matches.
`Iteration`	Wraps one component that has to occur at least as often as a defined minimum and as most as often as a defined maximum occurrence. Matches as long as the number of times its component matches the proceeding input is within the specified range of occurrences.
`Completion`	Is used within sequences to match all bytes up to the position from which the subsequent component in the sequence matches.

Model the Parse Tree The parse-tree is

`Capture`	Wraps one component and associates it with a name. This names the rule component (for reference) and the resulting parse tree node at the same time. As long as the wrapped component matches an element a frame is pushed onto the parse tree stack describing start and end position, nesting level and rule of the matching component.

Bootstrapping

`Reference`	Names the rule that this place-holder rule is substituted with when building the grammar. This allows to compose grammars programmatically and build rules having circular references to other rules. All references are replaced before a grammar is used. At runtime rules of type do no longer occur.

1.3 Terminal and Non-Terminal Rules

1.4 White-space

2 Parser as Data Controlled Solver

2.1 Index Overlay Parse-Trees

A parse-tree is modelled as a list of blocks. Each block results from a `Capture`. When processing input a `Block` "frame" is pushed onto the parse "stack" when it starts. As a result the list of blocks contains the root as its first element.

```
type ParseTree = [Block]
data Block = Block {
        start :: Position,
        end :: Position,
        level :: Level,
        rule :: Rule
}
type Level = Word8
```

Each Block memorises the absolute `start` and `end` `Position` of the block in the input (byte offset), the nesting `level` (starting from 0 for the root and increasing towards the leafs) and the `rule` that is captured.

The level is used to traverse the tree. For example in order to go to the next node all blocks with a higher level are skipped. The first block with the same or higher level as the starting one is the successor.

Note In most languages the `Block` structure is better implemented as multiple arrays. That is one for all starts, ends, levels and rules where values at the same indexes are one logical block.

3 Lingukit Grammar

```
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'
```