Formal Grammars as Data and Solvers

Jan Bernitt

May 11, 2014

Abstract

Specification for a programming language independent system of modelling a formal grammar as data and a formal process for a universal, formal grammar independent parser that builds index overlay parse-trees.

This allows to develop compilers purely by a grammar description and a interpreter for the resulting parse-tree. Parsing is decoupled from later analysis and emitter steps using pure data.

1 Formal Grammar as Data

Any 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 literally a set of production rules. Each production rule is composed out of `Rule` components. There is a fixed set of sufficient kinds 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 code-points of `UTF-8`. The non-terminal rules `Sequence`, `Selection`, `Iteration`, and `Completion` describe the nesting or structure of the syntax. A `Capture` decoration rule is used to name a rule for reference and as a node having that name in the resulting parse-tree. Finally the `Reference` allows the reuse of named rule components.

1.2 Kinds of Rules

Matching Bytes

`Literal` Matches an exact sequence of UTF-8 bytes. Typical examples are keywords of the target language.

`Terminal` Matches ranges of UTF-8 code-points (any single character of the input within the range).

`Pattern` Matches an abstract pattern of UTF-8 bytes. The length of matching bytes is given through a function for a particular pattern. A pattern can match none, one or more bytes or code-points. This is the only building block that cannot be interpreted other than invoking the pattern function. Patterns are mostly used to more efficiently match common sequences like white-space.

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` Decorates another rule and matches as long as the decorated rule matches at least as often as the specified minimum and at most as often as the specified maximum of occurrences.

`Completion` Is used within sequences to match up to the input position from which the subsequent rule component in the sequence matches.

Reference and Parse-Tree

`Capture` Decorates a rule and associates it with a grammar unique 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 this type do no longer occur.

2 Parser as Data Controlled Solver

The universal parser is a solver that is controlled by the production rule data of a grammar. This way it interprets input documents and produces a index overlay parse-tree.

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 Breaking with Conventional Wisdoms

3.1 Terminal and Non-Terminal Rules

While there are kinds of rules that could be said to be terminal rules and other that could be said to be non-terminal ones this classification is neither necessary nor particular helpful as it is orthogonal to the concerns one has to reason about. Therefore this terms are avoided widely throughout the description. All rules are of the same type but different kinds. If there are groups to distinguish those are the kinds that match bytes or code-points and those that match the structure of the syntax.

3.2 White-space

White-space is considered to be meaningful and important content as it one of the variable parts of most syntaxes that humans do care about a lot. It is not stripped out and has to be matched explicitly. It is not different at all from any other content. A grammar's author can control whether or not it is captured in the same way other content is captured. Special `Pattern` rules are used to make matching white-space convenient again.

3.3 Encoding

The system of grammars as data and solvers could easily applied to any character encoding but it is the belief of the author that encodings are a major source of accidental complexity and have shown to be a common source of software defects that could be avoided.

The `Unicode` standard might not be perfect but it is sufficient for universal usage. Out of the available `Unicode` encodings `UTF-8` balances the different objectives better than others and is already widely adopted. To emphasis the importance to use the same encoding everywhere the system is described particularly for `UTF-8`. A flexibility in the use of encodings is not a feature but a burden for both programmers and users.

3.4 Code Generation

Many of the popular parser-generators complect the grammar declaration with parse-tree or AST concerns. Code fragments are attached to the grammar definitions from which parser and AST code is generated. This clearly intermingles the concerns of parsing with those of analysis and code emitting and makes it impossible to share the same grammar declaration independent of the language the compiler is written in.

The seconds major drawback is that a code generation step hinders REPL-like feedback when working with the grammar. Even thou many tool have spent huge efforts in making grammar development smoothly none of the code generation tools could conveys a instant and interactive experience.

Furthermore code generation renders direct runtime integrations impossible. Languages like Smalltalk would e.g. allow to have changes to the grammar directly effect the whole environment.

3.5 Keywords

There is no special notion or additional declaration for keywords in the system of grammars as data. This allows a grammar alone to be sufficient as a complete working definition of a language's syntax and parse-tree structure. As a consequence pure data can be exchanges e.g. in files to control solvers according to a language. For example code highlighters could be build once and feed with more languages from a grammar file alone.

4 Lingukit

Lingukit is a concrete formal grammar to describe grammars. It is close to the `Rule` based model and has concise declarative syntax similar to BNF.

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'
```