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 Language as Data

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. The three terminal rules Literal, Terminal and Pattern match bytes of UTF-8. The non-terminal rules Sequence, Selection, Iteration, and Completion describe the nesting or structure.

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

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

`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.

2 Parser as Data Controlled Solver

3 Lingukit Grammar