lingukit

easy understandable parsing

(this document is available as PDF or HTML for printing, source is markdown)

Self-Experiment

1. Pretend you don't know anything about parsing (theory/practice).
2. Imagine a language!
3. Think: How would one read and understand its elements or structure?
4. Picture how you could formalise this to let a machine do what you just did in your mind.

Now, was any of the madness you **do** know about parsing relevant in this?

Imagining and design languages is real fun - making a machine parse it in the reality of *common wisdom* parsing however often is exhausting if not frustrating.

I chose to ignore and *forget* this wisdom and explore parsing once again.

A Journey from Grammars to Parse-Trees We are willing to write a grammar so we can use a parser that gives us parse-trees for input source files.

```
(Grammar -> Parser) -> Source -> ParseTree
```

That's basically how we used to think of it - because we made the assumption that we need a specific parser for a specific grammar. While this appears *natural* it is a fallacy. It is enough to have **one parser** that given a grammar can parse a source in accordance with the grammar.

```
Parser -> (Grammar, Source) -> ParseTree
```

Now we *only* have to write a grammar that we as well as the parser do understand. Although we also need to understand how the parser processes the grammar so that we can make it do what we meant to say. This sounds a lot like giving instructions - doesn't it? (*see also:* Damian Conway - Everything You Know About Regexes Is Wrong)

Following that thought a parser is sort of a *virtual machine* that - given *instructions* in form of a grammar - can process input *programs* what produces parse-trees as output.

Fortunately we are used to programming, to give instructions and reason about their implications and conditions. As soon as we have learned the *machine's primitives* a grammar can be written like a program. No special parsing theory or tool behaviour knowledge is required following this path.

Examples & Reference Implementation

See in the `examples` folder for actual grammars. There is a reference implementation in Java. The Dart implementation has not been updated lately with the latest changes.

Instructions

The following describes 8 essential and 3 optional instructions of a underlying parser's *machine language* through a surface syntax called `lingukit`, that maps roughly 1:1 to *machine instructions*.

Executing a instruction results either in a new *matched* input position or a *mismatch* position. The *execution context* is the current position and the parse-tree build so far.

Matching Characters

`lingukit` is designed to match bytes (of UTF8 characters) but the principle can be applied to any *basic unit*.

---

0 Literal Tests if input at current position literally matches a constant of one or more bytes given as a string literal with single quotes `'`.

```
'a' 'what we are looking for'
```

Alternativly a unicode character can be given by its code-point value in hexadecimal notation prefixed with `U+` (here `\n` what is decimal 10):

```
U+000A
```

Instruction Pseudo-code:

```
if (input starts-at position == literal)
    continue at position + length(literal)
else
    mismatch at position
```

---

1 Terminal Tests if input at current position is contained in a set of Unicode code-points. Sets are given in curly braces by single characters and character ranges:

```
{'a' 'b' 'c'} {'a'-'z'} {U+0041-U+005A}
```

Of course literals used in set definitions must be single code-point literals. Sets can also exclude characters using `!` *NOT* in front of a character or character-range in from `{low}-{high}`.

```
{!'?'} {!'0'-'9'}
```

There are also a couple of short hands (see syntactic sugar) that are not quoted. For example `Z` is the short-hand for `{'A'-'Z'}`. Such short hands or named sets (as described in capturing section) may as well appear within another set definition what is combined with *OR* usually or as *AND* in case of exclusion set.

```
{ Z '0'-'9' }
```

Instruction pseudo-code:

```
if (character-set contains (input code-point at position))
    continue at position + length(code-point at position)
else
    mismatch at position
```

Matching *Words*

---

2 Sequence A sequence of instructions is - surprise - given by writing the instructions one after another (separated by white-space where ambiguous otherwise).

```
'a' 'b' 'c'
```

Parentheses can be used to group sequences for nesting structures.

```
('a' 'b') 'c'
```

If one instruction in a sequences results in a mismatch the sequence results in a mismatch as well.

Instruction pseudo-code:

```
cursor = position
foreach instruction in sequence
    cursor = instruction exec (input, cursor)
    if (is-mismatch(cursor))
        mismatch at position
continue at cursor
```

---

3 Iteration Executes an instruction several times (between a minimum and a maximum). The iteration count is directly appended to the repeated instruction using `x{min}-{max}`. Some examples:

```
'a'x1-2 'b'x4 ('c' 'd')x3 {'0'-'9'}x2-4
```

Instruction pseudo-code:

```
match = position
do maximum times
    cursor = instruction exec (input, match)
    if (is-mismatch(cursor))
        if (done less than minimum times)
            mismatch at position
        else
            continue at match
    else
        match = cursor
continue at match
```

---

4 Selection Tests a sequence of alternatives until the **first match**. If no alternative matches the selection is a mismatch. Note that this is not a logical *OR*, the first matching instruction is continued, the sequence of alternatives is relevant. This is important to be able to reason about what will happen for a certain input sequence.

Alternatives are separated with the vertical bar `|`.

```
'ab' | 'cd'
```

Instruction pseudo-code:

```
furthest-mismatch = mismatch at position
foreach instruction in sequence
    cursor = instruction exec (input, cursor)
    if (is-match(cursor))
        continue at cursor
    else
        furthest-mismatch = furthest(cursor, furthest-mismatch)
mismatch at furthest-mismatch
```

---

5 Completion Consumes the input until the completed instruction matches at the current position. So instead of describing what to match the input is processed until a specific end is found matched through any another simple or composed instruction.

A completion is indicated by two dots `..` followed by the end instruction. Here an example to match XML comments:

```
'<!--' .. '-->'
```

Instruction pseudo-code:

```
end = length(input)
while (position < end)
    cursor = end-instruction exec (input, position)
    if (is-mismatch(cursor))
        position = increment(position)
    else
        continue at position
mismatch at end
```

Completions can be *expensive* in case the end-instruction is not a a literal or terminal as the position is incremented one by one.

Capturing Matches

Instructions 0-5 control the parsing process by instructing the parser what or how to match input.

The next two instructions 6 and 7 are used to a) shape the resulting parse-tree and b) allow to form reusable compositions and recursion.

6 Reference Combinations of instructions are *assigned* to a named rule.

```
comment = '<!--' .. '-->'
```

Such rules can be referenced on the right hand side of another rule through their name.

```
xml = comment | element
```

The rule `xml` *reuses* the rules `comment` and `element` as alternatives of a selection.

References can always be resolved before a grammar is actually used to control a parser. In practice they might just be used to initially describe recursion and reuse in a grammar through a instruction. Later on they might not appear any longer in an actual grammar instance (the runtime representation) as they have been substituted with the actual referenced rule. However, this can be implemented either way.

Instruction pseudo-code (when resolved during parsing):

```
referenced-instruction = context resolve reference-name
continue at referenced-instruction exec (input, position)
```

---

7 Capture Records the start and end position of the captured **rule instruction** by pushing a frame onto a stack (a the parse-tree in a sequential form).

Rules (simple or complex named instructions) also serve for the purpose to *mark* a section of interest that should be reflected in the resulting parse-tree. By assigning instructions to a identifier the given combination of instructions is of interest to us and will be represented in the parse-tree by a node with the identifier given.

```
identifier = {instructions}
```

A matched section becomes a stack frame (tree node) describing the rule that matched (most importantly its identity/name) as well as the start and end position of the match. The stack will itself add nesting depth information so that a sequence of frames can be looked at as a tree.

If combinations of instructions are not of interest as a complete element but should be used as a fragment to be able to reuse partial structures a reference to a fragment is prefixed with the minus `-` sign.

```
upper = {'A'-'Z'}
lower = {'a'-'z'}
first-name = -upper -lower+
last-name = -upper -lower+
full-name = first-name >> last-name
```

As `upper` and `lower` are prefixed for `first-name` and `last-name` rule they will not appear in the result tree. `first-name` and `last-name` on the other hand will become tree-nodes as they are referenced by not prefixed. The resulting tree for `full-name` will have the structure:

```
full-name
    first-name
    last-name
```

Instruction pseudo-code

```
stack open frame (rule, position)
end = captured-instruction exec (input, position)
if (is-mismatch)
    stack pop
else
    stack close frame at end
continue at end
```

Additional Instructions

There are 3 more instructions that are not essential for the concept to work but that can improve and extend its functionality.

---

8 Pattern Patterns are abstract basic units. The instructions asks a pattern how many bytes at the current input position are matching.

`lingukit` has a fixed set of patterns exclusively used for processing white-space but the principle could be applied for any purpose. The patterns used by `lingukit` are all expressible through the essential instructions what ensures interoperability also for those parser platforms that do not support patterns at all.

- Indent: `>` = `{' ' \t}*` (may be indenting white-space; on same line)
- Separator `>>` = `{' ' \t}+` (must be indenting white-space; on same line)

- Gap: `,` = `_*` (may be white-space)
- Pad: `;` = `_+` (must be white-space)
- Wrap: `.` = `>> \n >>` (must be line wrap)

Instruction pseudo-code:

```
length = pattern length-of-match(input, position)
if (length >= 0)
    continue at postion + length
else
    mismatch at position
```

Patterns are mostly a performance optimisations as almost all could similarly be modelled using combinations of other instructions.

Obs!! Different parsers might support different sets of named patterns so they should be used with caution. For the same reason Regexes should not be included as different platforms have different support and interpretation of regular expressions what would undermine the interoperability of the parser/grammars.

---

9 Decision This additional instruction is used as an element in a sequence to mark the position in that sequence where it is clear that the sequence is meant and should fully match. If the sequence matched up to the decision but mismatches at a later point the parsing has failed, no other alternatives should or have to be tried. The syntax error is at the mismatch position in the sequence.

A decision position is marked with an ampersand `&` sign. In a JSON grammar the `array` rule might be given as:

```
json ::= object | array | bool | null | string | number
array ::= '['&, -elements?, ']'
```

An `array` is the second alternative for a `json` element. If the element starts with an square bracket `[` we can be sure it has to be an array and the rule should fully match, otherwise the JSON is malformed.

Instruction pseudo-code (modification only):

A simple way to implement the instruction is to modify the machine behaviour of the sequence instruction. The *mismatch* path throws an exception in case the sequence had a decision and has been matches beyond it.

```
    if (sequence decision-index < current-element-index)
        raise exception "parseing failed"
    else
        mismatch at position
```

The decision instruction is a very useful instruction to be able to give helpful feedback in case of malformed input. Especially recursive data formats like JSON will otherwise often result in bad feedback as all alternatives will be tried first before the parsing fails whereby the parse-tree will be reduced to nothing when the mismatch travels up the parser's call stack.

---

10 Look-ahead The described parser *machine* has no different parsing modes as described so far as greedy/non-greedy is a source of confusion and unintuitive complexity.

Especially grammars for already existing languages might be ambiguous in a way that can just be resolved when looking *ahead* in the input stream without actually processing it as usual. The look-ahead instruction does this. It is used as last element in sequences to describe how we expect the input to continue to make the sequence match but without counting that tail as part of the match for the sequence.

Look-ahead is a group prefixed with a tilde `~( {look-ahead-instructions} )`.

```
    a = ('a' 'sequence' 'continues' ~('with' something) )
    b = ('a' 'sequence' 'continues' ~('with' nothing) )
```

As the start of both `a` and `b` is identical we cannot tell which of them we should choose. The look-ahead can than be used to distinguish them without *consuming* further input.

Instruction pseudo-code (modification only):

The instruction is also easiest implemented by modifying the sequence behaviour of the parser machine. Everything that has to be done is to check at the end of the loop iterating over the elements of the sequence if the currently processed instruction was of type *look-ahead*. In that case the end position is not updated but the the flow directly continues with the end position thus far (before the look-ahead instruction).

```
    if (current-instruction type == look-ahead)
        continue at end
    else
        end = cursor
```

Comments

`lingukit` allows to write comment lines. A comment line has to start with the percent sign `%` and ends at the end of the line.

```
% a comment
```

Comments have no further function or consequence than to add some prose to a grammar file.

Syntactic Sugar

White-space Characters As there is no escaping for quoted literals white-space has to be defined using the code-point syntax. For better readability there are the following short-hands (note that no quotes are used as for literals!):

- LF (Line Feed): `\n` = `U+000A`
- CR (Carriage Return): `\r` = `U+000D`
- HT (Horizontal Tabulation, tab) : `\t` = `U+0009`

Character Sets The `lingukit` syntax offers several short hands for commonly occurring character sets that can be used everywhere a set is valid.

- ASCII White-space: `_` = `{ U+0009 U+0013 U+0032 }`
- ASCII Letters (upper) = `Z` = `{'A'-'Z'}`
- ASCII Letters (lower) = `z` = `{'a'-'z'}`
- ASCII Letters (upper and lower): `@` = `{z Z}` = `{'a'-'z' 'A'-'Z'}`
- ASCII Numbers (hexa) = `#` = `{'0'-'9' 'A'-'F'}`
- ASCII Numbers (decimal) = `9` = `{'0'-'9'}`
- ASCII Numbers (octal) = `7` = `{'0'-'7'}`
- ASCII Numbers (binary) = `1` = `{'0' '1'}`
- Any Unicode code-point = `$` = `{ U+0000-U+7FFFFFFF }`

Implementation Notes

Components

Parser The universal parser is very straight forward to implement in all common languages. The full pseudo-code is given with each instruction. Everything is based on very basic programming constructs usually known and mastered already on novice programmer levels. Depending on the host language a parser might be written in about 30-200 LOC.

Instructions/Rules Instructions are basically data records or abstract data types (depending on the host language support) with no *own* functionality regarding the parsing process. Their task is to make instruction details available for the parser machine to interpretate.

Parse-Tree The parse tree is nothing more than a stack of `token` records of form:

```
token
    rule
    nesting-level
    start-position
    end-position
```

This is similar or known as *index overlay parse-tree*. The level allows to view a sequence of records as tree.

Performance I never measured performance but I know that pushing and pulling frames off and from the stack is easy to implement so that is boils done to a few array store operations and integer arithmetic. The parser itself will only require relatively small stack frames for each nesting of instructions in the grammar. Further heap allocation is not needed. No other mutation than the parse-tree stack takes place.

The costs of comparing input with expectations have to be paid in any parser technology but the slim process that does all work in one step keeps this almost as essential as possible.

The way parsing works also implies that neither large grammars nor huge input affect the parsing in a non-linear manner. For the most part these do not matter. It should also be mentioned that in contrast to *common wisdom* parsing `lingukit` grammars will necessarily be written so that the first path that matches is taken (independently of the questions if other alternatives might match as well). This should keep mismatching alternatives short on average.

Tweaks & Optimisations Grammars are instruction trees, a data structure that can be analysed and optimised before it is used. Rules can be rewritten/replaced with simplified ones that will have the same behaviour. The most trivial example is to fuse multiple literals following each other in a sequence into one longer literal. The goal always is to reduce the nesting and size of the tree as smaller trees will result in less function calls, thus less stack frames and branching.

In principle this also enables to formulate grammars in inadequate ways (e.g. using left recursion) as long as a rewriting procedure is known that transforms the instruction tree to a adequate one with the intended behaviour.

Extensions & Modifications The core idea is to use instruction trees and a parsing *machine* interpreting these. In principle both the essential as well as the optional instructions chosen for `lingukit` so far are not special in some way - they just were obvious and useful to me. Each of them could be removed, other instructions (not described or thought of here) could be added.

Tooling As grammars are data general tools can be build around them, interpreting their tree structure e.g. to convert it to other formats like a visual graph.

Non-Textual Implementations `lingukit` is a somewhat BNF like syntax in which grammars are described as text that is parsed to tree from which a runtime grammar is build.
This additional step might not be worth it in some languages. In particular functional languages like Haskell or lisps are well suited to directly *formulate* a grammar's data structures in the host language itself. This approach is also used to bootstrap a parser for the `lingukit` language.

Q & A

Can all languages (syntaxes) be expressed?

No. Selection sticks to the first match. When using the same rule nested within different parents and expecting them to match different options for the same input sequence depending on the parent or grant parent etc. will not work. Using look-ahead might help to fix some ambiguity shortly after the matched selection options but in general the grammar has to be written so that the first matching option is what we want. If this does not do it cannot be said.

How to resolve ambiguity in syntaxes, give precedence to a rule?

Remember that a selection goes with the first option that matches. To give an option a higher precedence move it to occur earlier in the selection. Or in other words sort options by their precedence starting with the highest. Naturally it is a good idea to design a syntax so that most of the options start with a literal that is different from the other options first literal. Also it is good practice to have options starting with a literal first, options with variable start last.

Can the grammars be declared using left-recursive/right-recursive rules?

Currently any left-recursive grammar will cause the parser to eat stack until a stack overflow occurs. Thus recursion may never occur as first element of a rule sequence. Some input has to be consumed before recursion occurs to make sure the parser makes progress.

Won't right recursion cause problems with the stack, a *large* parser?

Classic grammars had no iteration construct so theoretically endless lists of elements had to be modelled using left or right recursion where in case of right recursion each recursion would create another stack frame. However, `lingukit` has an iteration instruction that allows repetition of rules without using the stack, thus it is not a problem.

How to make white-space significant or captured in the parse-tree?

The described parser machine has no lexer or lexing phase. White-space is as good as any other input, that is to say it is always significant and captured in the same way other input is or is not. This is why most grammars heavily use `,` as it means *any white-space*. As this syntactic sugar has no name it is not captured by default. If white-space should be captured it just has to be named like `(,):white-space` would do.

How to describe or distinguish terminal/non-terminal tokens and fragments?

To distinguish terminal and non-terminal tokens might be theoretically interesting but is of little importance when writing grammars. Rules without a reference are terminals, the rest isn't. However it is of far bigger practical interest to distinguish rules that form *words* (no white-space between the parts) and rules that form *sentences* (white-space between the parts). Now, as white-space is explicit in `lingukit` there isn't really a difference here. Use white-space literals or syntax sugar to gobble it between the words of a sentence or not use it between the letters of a word.

To use something as a fragment a reference is prefixed with minus `-` so that the referenced rule will not be captured itself. Note however that references in such a rule again have to be prefixed to not be captured. Otherwise the parser will assume we are interested to have a node for them.

Can the parser benefit from memoization?

Most likely not.

Can the parser be parallelised (use multi-threading)?

Not really. Parsing is a inherently sequential problem. Theoretically the flow could be forked on selections to compute each in parallel to faster find the first one matching throwing away everything else. But as they push and pop on and off the stack while they parse this would also require to given each thread its own stack merging the one of the matching option back to the main stack. One is most like better off with starting multiple parsers for different input files in parallel threads each parsing single threaded.

What more?

I later discovered Parsing with Derivatives by Matthew Might from Stanford University having ideas for parsing that seam to be related to me but tackling it from the theoretically point of view with a lot of mathematical yadda yadda yadda.