Scanning and Parsing Overview

A DSL is best built as a hierarchy.

At the "leaf nodes" of the hierarchy are characters (or better yet, glyphs[1]).

Leaf nodes are collected into composite structures, called *tokens*.

A language (and a DSL) is composed of layers. At each layer, a certain set of tokens is parsed and transpiled into another set of tokens.

An *unparser* is a layer processor. It takes a set of tokens and emits them as a simpler set of tokens. For example, we might construct a tree representing a Scheme program, then unparse it into a JavaScript program.

Scanner

The first layer of a DSL is the *scanner.* It reads input data, e.g. characters, and converts the data into *tokens*. A *token* might contain file and position information.

The scanner might compress runs of similar characters into single tokens. For example a run of whitespace – spaces, tabs, etc. - might be compressed into a single *whitespace token*. These are semantically uninteresting, but *whitespace tokens* can act as delimiters between other tokens.

More importantly, the scanner compresses semantically interesting runs of related characters. For example, an identifier, *id*, in a language is usually composed of a single alpha character followed by 0 or more alphanumeric characters. The scanner groups such identifier characters together and assigns a single token (e.g. a token code plus a string hashcode) to such runs. Compressing such runs of characters into single tokens relieves subsequent passes from having to re-scan the input character by character.[2] The scanner also worries about the low-level details of delimiter[3] characters. In languages

---

1 In fact, characters *are* glyphs. Characters are small pixmaps. The glyphs are bound to certain keys and/or certain input gestures, e.g. multiple keys pressed at the same time, mouse movement and buttons, etc. It is possible to imagine a wider set of glyphs, e.g. boxes and lines, that are not constrained to a fixed bitmap size. Historically, most programming languages are based on rigid, non-overlapping grids of character glyphs. This is due to the limitations of early hardware, keyboards, memory and processors. Newer hardware can display overlapping glyphs, but these advances are not widely used for programming language design, yet.

2 This used to be very important for efficiency. Wasting processor time in re-scanning character strings caused noticeable slow-downs in processing (and ate memory). Modern hardware processes character strings much more quickly, but the notion of *tokenizing* remains. On faster hardware, the act of creating tokens is not (necessarily) about speed, but does help with "divide & conquer". A scanner recognizes low-level strings, subsequent passes recognize patterns of increasing complexity – the *semantics*. Untangling scanning from semantics-recognition is easier to reason about when a scanner "gets the low-level details out of the way". See my other essays for more discussion about layers of pattern matchers.

3 Separator characters, such as ")", "(", etc.. For example, *ids* are terminated when a separator character is seen, e.g. when the next character is not an alphanumeric.

where indentation is significant (e.g. Python), the scanner might keep track of indentation levels and attach these to *tokens* or, might create special indentation *tokens*.
Scanners are, fundamentally, pattern matchers.

Scanners often use REGEX libraries to specify string matching.

A tool that build scanners from declarative specifications is LEX.

PEG technology obsoletes the use of LEX. See below.

Parser

A *parser* inputs a sequential stream of *tokens*, and checks that the tokens form valid phrases – in sequence – for the language being processed (compiled, transpiled, etc.).

Parsers are pattern matchers, albeit more elaborate than the pattern matchers in scanners.

Parsing is typically specified as a set of rules (patterns) that describe parts of the language being processed.

Parsing typically uses subroutines (stack based) to do pattern matching. As such, parser pattern matchers cannot use REGEX directly, since REGEX does not provide stack-based routines.

Parsers can match patterns across line boundaries, whereas REGEX cannot.

Parsers can match structured code (phrases), whereas REGEX cannot.

A tool that builds parsers from declarative specifications is YACC.

Parser fall, broadly, into two categories – top-down and bottom-up. Top-down parsers are typically implemented as recursive descent routines. Bottom-up parsers match phrases, then bubble such matches upwards to recognize a full program. Bottom-up parsers have been formalized (see the Dragon Book).

One such formalization is the set of languages called LR. A very common subset of LR is called LALR(1). YACC builds LALR(1) parsers, as state machines, from declarative specifications. Languages that fit the LALR(1) mould are a subset of LR languages. LR languages are a subset of more general languages. The restrictions of LR and LALR(1) languages are tolerated because they can be formalized. LALR(1) is popular because of the YACC tool.

Top-down languages are less restrictive than LR languages, but, top down parsers were not easily formalized.

LALR(1) (through YACC) was “first to market” and has gained most of the mind-share in this problem space.

Bryan Ford[4], in the early 2000’s, invented the PEG parsing technology, which creates top-down pattern-matchers (parsers) using a syntax similar to that of REGEX. PEG also uses backtracking in some cases.

4 https://bford.info/packrat/

The class of languages that PEG can recognize is larger than the LR class of languages. PEG can match some patterns that LR cannot match, and vice versa.

In my opinion, top-down parsing (e.g. PEG) matches languages that are easier to create than the class of LR languages.

In top-down pattern matching, order matters – the programmer can specify which patterns to match first. In LR, it is possible to create ambiguities which are, in my opinion, "not obvious". Such LR ambiguities need to be dealt with by rearranging the grammars or by using exception-processing rules. Typically, a LALR(1) grammar programmer waits for the tool to detect and declare ambiguities, instead of intuiting them.

PEG subsumes scanning and parsing into a single notation.

In my opinion, PEG is a break-through, which makes building DSLs very simple. I expect that simple DSLs[5] will over-take language design. It is now possible to design and use multiple DSLs in a single project.[6]

Tools such as OMetaII and Ohm-JS are based on PEG technology.

PEG libraries exist for many common languages, e.g. Ohm-JS for JavaScript, ESRAP for Common Lisp, etc.

S/SL

PEG is not the first tool to process and create top-down grammars. S/SL is another such tool. S/SL was invented in the 1980's[7].

S/SL has the advantage that it works with *tokens*, whereas Ohm-JS works only with character streams.

S/SL has the advantage that it is pass-based. Ohm-JS does not default to this behaviour, but could be twisted to conform to passes[8]. Since Ohm-JS accepts only characters as input, it discourages pass-based and token-based approaches.

S/SL has the disadvantage that it tends not to subsume scanning and parsing. PT Pascal[9], though, uses a scanner created in S/SL.

TXL

TXL[10] is a source-to-source technology, based on backtracking, ostensibly used for experimental language design.

---

5 Especially source-to-source translation.
6 See my essay "DSLs – The Future of Computing"
7 https://en.wikipedia.org/wiki/S/SL_programming_language#:~:text=The%20Syntax%2FSemantic%20Language%20(S,University%20of%20Toronto%20in%201980.&text=S%2FSL%20is%20designed%20to,syntax%20error%20recovery%20and%20repair.
8 See my essay "Ohm In Small Steps"
9 https://research.cs.queensu.ca/home/cordy/pub/downloads/ssl/
10 https://www.txl.ca/

PROLOG

PROLOG is a language that expresses exhaustive pattern-matching using backtracking.

As such, PROLOG should be ideal for building parsers (and, maybe, scanners).

The early wisdom stated that backtracking was not a viable strategy for building parsers. Early attempts to use backtracking in parsers was quashed after Early's (a researcher's name) Parsing technology.

Now, with much faster computers, and almost-unlimited memory, this "wisdom" could be re-investigated. To my knowledge, this has not been done.

PEG parsers use a limited form of backtracking.

Searches for parsers built in PROLOG tend to turn up experimental research into natural language parsing (a superset of PEG based languages and LR languages).