E-Expressions

Okuno Zankoku

January 13, 2021

I'm *so done* writing parsers for my language experiments. I always want a whole host of ergonomic features, decent speed, and good error reporting, but it always takes *so long* when what I want to be doing is playing with semantics. So, I resolved to write one last parser which can be re-used across all my experiments. This parser would deal with all of the character manipulation, leaving the question of concrete syntax to one of simple pattern-matching against an abstract data type. Thus, I implemented "ergonomic expressions" (or maybe "elaborate"? "extended"?) or "e-exprs"[1] for short.

Why not just s-expressions? Don't get me wrong, my father taught me Scheme last century; it was my first programming language, and to this day I always install Racket on new computers because I never know when I might want to play. Nevertheless—controversial take—I think Lisps have too many parenthesis,[2] and that is a sign of a too low-level interface. A user-facing syntax is a user interface, and as such, it should have basic amenities. Lisp intensionally lacks ergonomic features because its users largely believe that s-expressions are so beneficial to macro-writing that it outweighs the cost of the harsh ergonomics. More recent languages show that any language that supports abstract data types can manipulate its own arbitrarily complex syntax as easily as a Lisp: the benefit is wholly imaginary, and therefore cannot be worth even a small cost.

Overall though, e-exprs are not so different from s-exprs—there are just more ways to combine them into larger expressions. Let's quickly go over the distinctive features of e-exprs.

[1] I'll let you decide how to pronounce that `:)`, but I like "*eks*-per"

[2] Also, Lisps use parenthesis in a strange way: they indicate function application but never grouping, whereas mathematical practice, and especially when working with the lambda calculus, is to use parentheses mostly for grouping. Of course, many languages, and even standard mathematics, use them for both grouping and function call, but Haskell and ML don't.

1 A Whirlwind Tour

Perhaps the largest conceptual distinction is that combination is not always function[3] application. Parenthesis are conceptualized as grouping, though a specific language may interpret them differently. To that end, there are two other simple means of grouping: square brackets and curly braces. Unlike some proposed s-expr extensions, parens, brackets, and braces are kept distinct.

Ironically, s-exprs use a lot of line noise to build lists. I therefore included the ability to group expressions by comma-separating them.

```
'((f x) (g y) z)
```

```
[f x, g y, z]
```

Adding a leading or trailing comma doesn't change anything. You can even have a comma-separated list with zero items `(,)` or one item `(a,)`/`(,a)`. Semicolons work the same way, and add a second layer of grouping. Colons are similar, but separate exactly two subexpressions from each other. An ellipsis (spelled with two dots: `..`) may separate two expressions, but wither side may be empty.

Another way to combine expressions is to "chain" them together with dots, just as you would in almost any programming language for accessing a field of a record.

```
(point-x (player-position (world-player world)))
```

```
world.player.position.x
```

Unlike other languages, you can chain any two e-exprs together with dots. Additionally, the dot is optional if the next expression in the chain is surrounded by a bracketing form (parens, square brackets, curly braces, or an indented block, which we will soon see). This means that syntaxes that look like field access, array indexing, or function application are represented by the same, unified syntax.

```
matrix[2, 3]
point3d{x, y}.scale(4)
```

[3] or special form, or macro

Bear in mind that the lack of whitespace around the dot is critical: adding whitespace changes the meaning, as we will see later.

The most visually-apparent change is the addition of indentation-sensitivity. Importantly, an indented block is always preceded by an end-of-line colon. One of the most iconic Lisp notions, `cond` looks terrible, but applying indentation-sensitivity, chaining, and colon-separation can clean it up significantly.

```
(cond
  ((p? a) (foo a))
  ((p? b) (foo b))
  (else foo c))

cond:
  p? a: foo a
  p? b: foo b
  else: foo c
```

An indented block can also appear on its own rather than as part of a chain. Note the space before the colon that ensures the block isn't chained with the `z` argument.

```
def big_function x y z :
  xy = combine x y
  z' = transpose z
  z' * xy
```

Remember that there isn't much of a denotational semantics for e-exprs; they are pretty much exactly what you write. In the case of the last example, we simply have a sequence of expressions, the last of which is a block containing three subexpressions, each of which is just a list of symbols. Although it appears we are using binary operators, e-exprs do not have to support them. A layer on top of e-exprs can be responsible for detecting and rewriting any infix—or indeed mixfix—operators, and indeed my e-exprs library supplies just such a system.

E-exprs also support a handful of other smaller features. String templates `"Hello, ``titlecase name``!"` are a feature I can no longer do without. Unicode symbols `λ x: x + 1` allow for more math-like notation if desired. Heredocs allow for large strings to be used as documentation, quick-and-dirty data blocs, or to embed other languages that a macro system can pick up and translate. Synthetic infix functions `this .(not a`

`real method) arg1 (arg 2)` allow for function call syntax to be re-ordered to express a more naturalistic reading.

E-exprs *don't* support any language-specific concepts like cons-dot, tick-quotation, quasi-quote/unquote/unquote-splicing, or reader macros. I point out the Lisp ideas specifically, since Lisp is often accused of having "no syntax", but e-exprs also don't have any concepts from more conventional languages. Even `[1, 2, 3]` is not a list—as far as e-exprs are concerned, it is a comma-separated sequence of expressions enclosed in square brackets. If you want that to be a list in your language then it can be, or it could be something completely different; in fact, as part of a chain, it might make sense if it were a multi-dimensional array access.

The point is that e-exprs don't solve the problem of concrete syntax. Instead, e-exprs deal with the messy character stream to produce a straightforward tree data structure containing widespread notations. Bridging the gap from an e-expr to an abstract syntax is much easier because such a tree is amenable to simple functional pattern matching. I hope to be able to take any formal programming calculus from the literature and have a e-expr based concrete syntax built inside of fifteen minutes—one which I will enjoy writing in so that I can really experiment with the semantic ideas.

2 E-expr Grammar

E-expressions are parsed from UTF-8 encoded Unicode text streams. Due to some context-sensitivity, the following schematic is how I will describe the overall parsing process:

$$\text{Unicode} \xrightarrow{\text{lexer}} \text{Pre-tokens} \xrightarrow{\text{post-lexer}} \text{Tokens} \xrightarrow{\text{parser}} \text{E-Expr}$$

The lexer can be described mostly in terms of regular expressions (with one exception involving a capturing group). The post-lexer analyzes constructs involving whitespace-sensitivity and simplifies the token stream. The parser, finally, is a context-free grammar over these tokens.

Remark. *I have decided to allow only Unicode codepoints in the alphabet and decided on UTF-8 as the only available encoding for now. Selecting Unicode seems the most future-proof choice, since its goal is to subsume all other character sets. Choosing UTF-8 as the only encoding is less reliable, but detection of magic strings in their appropriate encodings is a source of complexity. I may allow a file-initial comment stating encoding in the future, but for now I think UTF-8 will be an acceptable encoding for most program texts.*

To avoid the verbosity of academic regular expressions, we allow some notations that readers may not be familiar with. Let's take the opportunity to state our notation explicitly:

- Literal characters and strings are written in `typewriter` font and surrounded by `'single ticks'`. We use standard C-style character escapes for otherwise hard-to-typeset whitespace characters (space, tab, and newlines), as well as backslash itself; to be safe, we also backslash-escape single- and double-quotes and backtick.
- Character classes are written with colon-square-brackets [: ... :]. Negative character classes are preceded by a negation sign ¬[: ... :]. The contents of a character class are the union of the literal characters, character ranges, and named character classes appearing inside the brackets. We define some character classes; in doing so we use plus/minus for set union/difference. Character ranges are written as two literals separated by an endash `'a'–'b'` and stand for the set of all code points between the two literals (inclusive); this is not to be confused with set minus.
- Alternation is written with a pipe $r_1 \mid r_2$.
- An optional element is written with a superscript question mark $r^?$.
- Repetition zero/one or more times is written with a superscript star/plus respectively $r^\star / r^+$.
- Repetition by a fixed amount is written with that amount superscripted in curly braces $r^{\{n\}}$.
- Difference of regular languages is represented with a minus sign $r_1 - r_2$. Although regular languages are closed under subtraction, this notion has not made it into implementations. An input matches $r_1 - r_2$ exactly when it matches r_1 but does *not* match r_2.
- We use ω to match any single character, and ε for the empty string.
- Parentheses for non-capturing grouping.
- $r^{=x}$ captures the portion of input that matched r and stores it in the variable x for later use.
- When a regular expression becomes to long to be easily-understood, we will resort to the convensions of Backus-Naur Form. Non-terminals are typeset in **boldface**.

- We will need a couple families of regular expression parameterized by a string. These are written as if they were a function **family**(x).
- Where a variable appears free in the right-hand side of a rule, it is a unification variable.

2.1 Lexer

The first step in parsing an e-expression is to identify pre-tokens from a stream of UTF-8 characters. The grammars for most pre-tokens mostly describe regular languages, but we have made some use of BNF notation to help keep the regular expressions small and understandable. TODO: but... heredocs! These pre-tokens may need to later be classified into tokens proper

$$
\begin{aligned}
\textbf{preTokens} &::= \textbf{preToken}^\star \\
\textbf{preToken} &::= \textbf{encloser} \mid \textbf{separator} \\
&\quad \mid \textbf{preSpace} \mid \textbf{preNewline} \\
&\quad \mid \textbf{comment} \\
&\quad \mid \textbf{symbol} \mid \textbf{number} \mid \textbf{string} \mid \textbf{char}
\end{aligned}
$$

Enclosers and separators are baked into the grammar of e-exprs.

$$
\begin{aligned}
\textbf{encloser} &::= [\text{: '() []\{\}' :}] \\
\textbf{separator} &::= \texttt{','} \mid \texttt{'.'} \mid \texttt{'..'} \mid \texttt{';'} \mid \texttt{':'}
\end{aligned}
$$

Whitespace is significant in e-exprs in two ways. Most prominently, e-exprs are indentation-sensitive. Some pairs of tokens require whitespace between them, and some tokens change their meaning depending on the presence/absence of whitespace.

$$
\begin{aligned}
\textbf{preSpace} &::= [\text{: '\ \t' :}]^+ \\
&\quad \mid \texttt{'\textbackslash\textbackslash n'} \\
\textbf{preNewline} &::= \texttt{'\textbackslash n'} \\
\textbf{comment} &::= \texttt{'\#'}\ \neg[\text{: '\textbackslash n' :}]^\star
\end{aligned}
$$

Symbols are likely to make up most of the text in an e-expr, as they can be used for variables, operators, keywords, and so on.

TODO: I haven't decided if I really want to allow colons to start symbols.

$$
\begin{aligned}
\textbf{symbol} ::= {} & [: \mathit{symbolChar} :]^{+} - (\textbf{startNumber}\ \omega^{\star}) \\
& \mid \texttt{':'}[: \mathit{symbolChar}, \texttt{':'} :]^{+} \\
\mathit{symbolChar} = {} & [: \mathit{letter}, \mathit{digit}, \mathit{mathSymbol}, \mathit{currencySymbol} :] \\
& + [: \texttt{'\~!@\$\%\^{}\&*-\_=+|<>/?'} :] \\
& - [: \texttt{'\ \ \textbackslash t\textbackslash n\textbackslash r\#\textbackslash\textbackslash'} :] \\
& - [: \texttt{'()[],.;:\textbackslash`\textbackslash'\textbackslash"'} :] \\
\textbf{startNumber} = {} & [: \texttt{'+-'} :]^{?}\ [: \mathit{digit} :]
\end{aligned}
$$

Numbers can be represented in base 2, 8, 10, 12, or 16; succinct definition requires the use fo parameterized rules. The base prefix (**prefix**) and special exponent (**exp**) codes are as you would expect, so I've delayed defining them until later.

FIXME: underscores as digit group separators

$$
\begin{aligned}
\textbf{number} ::= {} & [: \texttt{'+-'} :]^{?}\ \textbf{prefix}(n)\ \textbf{digit}(n)^{+}\ \textbf{scientific}(n)^{?} \\
\textbf{scientific}(n) ::= {} & \texttt{'.'}\ \textbf{digit}(n)^{+} \\
& \mid \texttt{'.'}\ \textbf{digit}(n)^{+}\ \textbf{exp}(n)\ [: \texttt{'+-'} :]^{?}\ \textbf{digit}(n)^{+} \\
& \mid \texttt{'.'}\ \textbf{digit}(n)^{+}\ [: \texttt{'pP'} :]\ \textbf{anyInt} \\
\textbf{anyInt} ::= {} & [: \texttt{'+-'} :]^{?}\ \textbf{prefix}(n)\ \textbf{digit}(n)^{+}
\end{aligned}
$$

Characters and strings are also available literals.

$$
\begin{aligned}
\textbf{char} ::= {} & \texttt{'\textbackslash''}\ \textbf{charContent}\ \texttt{'\textbackslash''} \\
\textbf{string} ::= {} & \textbf{stringLiteral} \\
& \mid \textbf{templateInitial} \mid \textbf{templateInner} \mid \textbf{templateFinal} \\
\textbf{stringLiteral} ::= {} & \texttt{'\textbackslash"'}\ \textbf{stringContent}\ \texttt{'\textbackslash"'} \\
& \mid \textbf{heredoc}
\end{aligned}
$$

The lexer does not attempt to match up the various parts of a string template. Instead, it merely identifies the potential parts of a string template, classigying each as an initial, inner, or final part of the tamplate. The parser will ensure that these match up as part of bracket-matching.

$$
\begin{aligned}
\textbf{templateInitial} ::= {} & \texttt{'\textbackslash"'}\ \textbf{stringContent}\ \texttt{'\textbackslash`'} \\
\textbf{templateInner} & \\
::= {} & \texttt{'\textbackslash`'}\ \textbf{stringContent}\ \texttt{'\textbackslash`'} \\
\textbf{templateFinal} ::= {} & \texttt{'\textbackslash`'}\ \textbf{stringContent}\ \texttt{'\textbackslash"'}
\end{aligned}
$$

Most UTF-8 characters are available for use directly in characters and strings, with some obvious exceptions. Otherwise, a number of escapes are available to both overcome the illegal characters or for the author to direct attention.

$$
\begin{aligned}
\textbf{charContent} ::= {} & \neg[: \texttt{'\textbackslash"\textbackslash`\textbackslash r\textbackslash n\textbackslash\textbackslash'} :] \\
| {} & \texttt{'\textbackslash\textbackslash x'}\ [: \mathit{hexDigit} :]^{\{2\}} \\
| {} & \texttt{'\textbackslash\textbackslash u'}\ [: \mathit{hexDigit} :]^{\{4\}} \\
| {} & \texttt{'\textbackslash\textbackslash U'}\ [: \mathit{hexDigit} :]^{+} \\
| {} & \texttt{'\textbackslash\textbackslash'}\ [: \texttt{'\textbackslash\textbackslash 0abefnrtv\textbackslash'\textbackslash"\textbackslash`'} :] \\
\textbf{stringContent} ::= {} & \textbf{charContent} \\
| {} & \texttt{'\textbackslash\textbackslash\&'} \\
| {} & \texttt{'\textbackslash\textbackslash'}\ \texttt{'\textbackslash n'}\ [: \texttt{'\textbackslash\ \textbackslash t'} :]^{\star}\ \texttt{'\textbackslash\textbackslash'}
\end{aligned}
$$

Heredocs offer an alternative for embedding long strings into e-exprs. They can be multi-line and do not require escape sequences anywhere, but they also do not allow escape sequences or templating.

$$
\begin{aligned}
\textbf{heredoc} ::= {} & \texttt{'\textbackslash"\textbackslash"\textbackslash"'}\ ([: \mathit{alphaNum} :]^{*})^{=\mathit{delim}}\ \texttt{'\textbackslash n'} \\
& \textbf{hdLine}(\mathit{delim})^{*} \\
& \textbf{hdEnd}(\mathit{delim}) \\
\textbf{hdEnd}(\mathit{delim}) ::= {} & \mathit{delim}\ \texttt{'\textbackslash"\textbackslash"\textbackslash"'} \\
\textbf{hdLine}(\mathit{delim}) ::= {} & (\textbf{line} - \textbf{hdEnd}(\mathit{delim})\ \textbf{line})\ \texttt{'\textbackslash n'} \\
\textbf{line} ::= {} & \neg[: \texttt{'\textbackslash n'} :]^{*}
\end{aligned}
$$

The prefix codes and special exponent codes are only defined for a select few bases. All other bases can be treated as never-matching.

$$
\begin{aligned}
\textbf{prefix}(10) ::= {} & \varepsilon \\
\textbf{prefix}(2) ::= {} & \texttt{'0'}\ [: bB :] \\
\textbf{prefix}(8) ::= {} & \texttt{'0'}\ [: oO :] \\
\textbf{prefix}(12) ::= {} & \texttt{'0'}\ [: zZ :] \\
\textbf{prefix}(16) ::= {} & \texttt{'0'}\ [: xX :] \\
\textbf{exp}(10) ::= {} & [: eE :] \\
\textbf{exp}(2) ::= {} & [: bB :] \\
\textbf{exp}(8) ::= {} & [: oO :] \\
\textbf{exp}(16) ::= {} & [: hH :]
\end{aligned}
$$

For completeness, we have familiar digit classes for different bases.

$$
\begin{aligned}
\mathit{hexDigit} &= \mathbf{digit}(16) \\
\mathbf{digit}(2) &= [\!: \texttt{'01'} :\!] \\
\mathbf{digit}(8) &= [\!: \texttt{'0'}\text{–}\texttt{'7'} :\!] \\
\mathbf{digit}(10) &= [\!: \mathit{digit} :\!] \\
\mathbf{digit}(12) &= [\!: \texttt{'0'}\text{–}\texttt{'9'}, \texttt{'ↂↃ'} :\!] \\
\mathbf{digit}(16) &= [\!: \texttt{'0'}\text{–}\texttt{'9'}, \texttt{'A'}\text{–}\texttt{'F'}, \texttt{'a'}\text{–}\texttt{'f'} :\!]
\end{aligned}
$$

Remark. *I don't especially want to enter into the spaces vs. tabs religious war. I therefore allow both, even though tabs have implementation-defined behavior.*

Remark. *One religions war I will get involved in is LF vs. CRLF. I've only allowed the LF form of newline, because who wants to deal with OS-specific newline sequences in 2021? Perhaps your text editor can be configured, and perhaps your VCS is "smart", but why not agree to stop paying the complexity cost when all we get out of it is one company's being pampered? That said, as little as single open issue might change my mind.*

Remark. *I've chosen to not have block comments. Block comments are fragile unless they are able to nest inside each other. For historical reasons, most languages do not have nesting block comments, so I've found that text editors do not bother implementing good support for nesting block comments. The last thing I want is to ask a potential e-expr user to change their workflow, so I've simply ignored block comments altogether. Instead, your text editor likely has good support for commenting out all selected lines at once, which is all a block comment does. Besides, using only line comments makes changes more easily spotted when using line-based version control.*

Remark. *The set of symbol characters is* not *wisely chosen overall. I just took a peek at what Unicode had to offer, picked some reasonable-looking properties. Outside of the ASCII portion and some well-known mathematical symbols, I am open to critique, especially since I have done none myself. I hope for now that this is a reasonable enough set so as not to cause confusion, but I expect it will be important to reduce the confusion caused by similar-looking glyphs (e.g. at the moment capital A and capital alpha look the same, but will be treated differently in code). Nevertheless, you can see that I've tried to include as many symbols as possible that are available on US/UK keyboards (programming seems to have settled on this as a de-facto standard,*

and I don't have the knowledge to change it; if you have gripes about what characters on your local keyboards you would most like to use/avoid in code, please get in touch!) I have explicitly ruled out whitespace and comment-related characters (for obvious reasons), as well as a number of characters used as punctuation in e-exprs.

The usual C-style escapes are available, including `\`` for backtick, as well as `\e` for ASCII 27 (escape) since it finds some use in terminal interface programming.

Remark. *Strings are not normally allowed to cross line-boundaries because a missing close-quote can lead to poor syntax error diagnostics. A string can be explicitly broken across lines using a backslash immediately before a newline (TODO without a comment?); the rest of the string picks after the matching backslash on the next line. Note that breaking a line across a string does not insert a newline; this must be inserted explicitly.*

Remark. *The `\x`, `\u`, and `\U` escapes offer access to the entire range of unicode codepoints. While `\x` and `\u` offer access to limited ranges of these codepoints (for use in bytestrings, ASCII escapes, or when only the Basic Multilingual Plane is needed), `\U` allows any code point to be specified—even those outside of the Unicode range. This ensures that `\U` is future-proofed in the unlikely event of Unicode (or a supersceding standards body) extending its codepoints beyond `0x10FFFF`. However, it also means that some escape sequences may be parsed as "too long", as in `\U101234f`; thus the null escape `\&` was added so that unruly escapes can be safely contained e.g. `\U101234&f`.*

Remark. *I've included base-12/dozenal numbers basically just because I can. It's not like it interferes with any other syntax, and once an implementation has procedures for binary, octal, decimal, and hexadecimal, throwing in dozenal shouldn't be any extra stretch. Yes, I know I criticized the complexity cost of CRLF, but... $wh im_{se} y$! Well, I might say I'm proving a point about how high-level languages can accommodate notions of number that are significantly divorced from the low-level details of computing hardware, but that's boring!*

FIXME: shouldn't I allow **symbol** ::= `'\\'` [: *symbolChar* :]$^+$?

3 Token Stream Cleanup

After the lexer identifies tokens in the input Unicode stream, we are left with a stream of pre-tokens which must now be validated and normalized.

TODO

3.1 Parser

4 E-exprs as a System

TODO: location tracking, error recovery, the pattern-matching library, the various output formats, the mixfix algorithm