Report on E-Exprs

Eric Demko

The key contribution of this report is a grammar and parser for a data language with optimized ergonomics for encoding a wide range of programming languages. They key idea—which has been used to great effect since the dawn of high-level languages[1]—is to define the concrete syntax of one language as a subset of the concrete syntax of a previously-implemented data language. Thus, when implementing a new concrete syntax, one can re-use all of the work of converting a byte stream into a algebraic data type (ADT), and need only perform functional pattern matching to inject it into a type representing your abstract syntax. The rest of this report will define eexprs, explicate the implementation technique, and motivate/rationalize my design choices.

asdf

Motivation

I've lost count of how many toy language implementations I've given up on after having put not-insignificant time into the parser; I'm *incredibly bored of writing parsers*. So, I resolved to stop writing them. As Lisp programmers are already aware, code is data: one obvious technique would be to encode the abstract syntax in an existing data format. While JSON parsers are readily available, actually reading or writing a term is physically painful. Other common data languages like XML and YAML suffer a similar fate. I think anyone would much prefer s-exprs or e-exprs.

JSON

```
{ "_type": "apply", "function": "push"
, "args": [ "x", {"_type": "list", "elems":
  [ {"_type": "apply", "function": "f", args: [1]}
  , {"_type": "apply", "function": "g", args: [2]}
  ]} ]
}
```

E-exprs `push x [f 1, g 2]`

S-exprs `(push x (list (f 1) (g 2)))`

[1] specifically: since 1958 with LISP

YAML

```
!apply
function: push
args:
  - x
  - - !apply {function: f, args: [1]}
    - !apply {function: g, args: [2]}
```

XML

```
<apply>
  <function>
    <var>push</var>
  </function>
  <args>
    <var>x</var>
    <list>
      <apply>
        <function>f</function>
        <args><int>1</int></args>
      </apply>
      <apply>
        <function>g</function>
        <args><int>2</int></args>
      </apply>
    </list>
  </args>
</apply>
```

The most widely-known data languages are optimized for machine-machine communication, with only some consideration for humans to peek at it occasionally. However, writing source code is all about human-human communication, with only some consideration for the ability of a machine to read it. Both s-exprs and e-exprs are much better for this task.

Recognizer Technique with Sexprs

S-exprs are the main inspiration for s-exprs. The two languages share a number of common features, but sexprs are a much smaller language. Therefore, I'll go over sexprs first to illustrate the technique succinctly before demonstrating eexprs. Though I have tried to distill the essence of this particular advantage of sexprs, if you are already familiar with this technique, it still might seem like flogging a dead horse.

The key idea is to separate parsing for a target language into two stages:

1. Parse a byte stream into the abstract syntax (i.e. data type) of s-exprs.

2. Recursively pattern match against the s-expr value to construct a term in the target language's abstract syntax.

In the case of sexprs, we have a particularly small abstract syntax:

```
data SExpr
  = Atom Text
  | Combination [SExpr]
```

Although one might have an idea about what the values of this type mean (e.g. that combinations are function calls), sexprs need not be a carrier for only Lisp dialects. One could just as easily use sexprs to represent the abstract syntax of (say) Prolog or Algol.

To use s-exprs as part of a lambda calculus implementation, we need only define our abstract syntax,

```
data Lam
  = Var Text
  | Lam [Text] [Lam]
  | App Lam [Lam]
```

and then define a recognizer,

```
recognize :: Sexpr -> Maybe Lam
recognize (Atom name)
  = Just (Var name)
recognize (Combination (Atom "lambda" : params : body)
  = Lam <$> map recognizeParams params <*> map recognize body
  where
  recognizeParams (Atom name) = Just name
  recognizeParams _ = Nothing
recognize (Combination (func : args)
  = App <$> recognize func <*> map recognize args
recognize _ = Nothing
```

and with only this handful of code, we have completed an entire parser for the lambda calculus.

Admittedly, this recognizer does not report informative errors. Later, we will see a "grammar combinator" library for e-exprs that tracks context and can therefore support much richer error diagnostics.

The advantages of the technique are

- A high-quality s-expr parser and grammar combinator library need only be implemented once, and then it can be shared between multitudes of new languages.
- Most of the tedious and tricky parts of parsing are in the s-expr parser; the code that needs to be written for new languages is small and simple.

The question now is, if s-exprs are so great, why implement e-exprs? My view is that s-exprs, while they are far more ergonomic than JSON, are *not* ergonomic compared to common programming languages. A claim like that is certain to spark vocal argument, but to engage in it here would derail the main thrust of this paper. Instead, let's jump straight into describing e-exprs and delay any rant-like objects until later in the report. Hopefully, the advantages of e-exprs will become clear as we examine them.

E-Exprs

A Whirlwind Tour

The leaves of an eexpr are atoms such as numbers, symbols, and strings. A range of Unicode symbols[2] are also supported, and there is no distinction between different classes of symbols (such as operator, variable). In addition to C-style strings, there are also also be written sql-style, or as heredocs, which are equivalent in the abstract grammar. All other eexprs are simply various combinations of these leaves. FIXME move this to a more detailed place where I can include rationale: There is no special syntax for character literals.

```
# numbers
12
0xf00d
6.636e-34
# symbols
filename
if
>>=
λ
a+3Ω
# strings
"Hello!"
'wouldn''t''ve'
"""
long form
  text that
 is "uninterpreted"
"""
```

The simplest ways to combine eexprs is to simply write them next to each other separated by spaces, or to enclose them in parenthesis. An eexpr can also be

[2]How useful unicode input turns out to be is a different question. I've set my system up to be able to input many mathematical characters, but it's a lot of work to ask from users in general. Instead, I would like to offer tools to can translate eexprs between Unicode and ASCII representations; it would be up to language and library implementors to offer an ASCII alternative for each Unicode symbol they use.

enclosed in square or curly braces, or separated by commas or semicolons. Each of these enclosers and separators are treated distinctly by eexprs, but an eexpr-based language may choose to ignore distinctions. There are no restrictions as to which separators can be used with which enclosers, or even that either needs the other at all.

```
greet name
(a + b)
[1, 2, 3]
{a b}; c; d
```

C-style strings can be made into templates that embed further eexprs in backticks within the string. Unlike some other string template syntaxes, arbitrary eexprs can be spliced into a template.

```
"Hello, `name`!"
"`results.len` result`if results.len == 1 then "" else "s"` found"
```

Eexprs can also be separated by dots, just as long as there is no whitespace around the dots. To support more familiar syntax, the dot is optional before strings and enclosers.

```
# all dots present
player.[1].position
r.'\bwords?\b'.search.("some words")
# equivalents:
player[1].position
r'\bwords?\b'.search("some words")
```

Eexprs can also be combined in an indented block, which are indicated by an end-of-line colon. Most of the time, the indented block should be a space-separated child of the other eexprs on the starting line rather than a child of the last dot-separated eexpr. Thus, the space before an indented block is optional.

```
# no-nice-things version
do :
  thing-1
  thing-2
# equivalent, but more familiar syntax
do:
  thing-1
  thing-2
# explicit dot-expressions are still possible
do.:
  thing-1
  thing-2
```

Since indented blocks are often enclosed, an end-of-line open encloser implicitly starts an indented block:

```
# this natural syntax
do {
  thing-1
  thing-2
}
# is equivalent to
do {:
  thing-1
  thing-2
}
```

Two eexprs can also be separated by a colon. These separating colons bind more tightly than comma, which makes it easy to encode key-value dictionaries in eexprs. Note that we can use indented blocks to allow the omission of oft-forgotten end-of-line commas.

```
{
  type: "point"
  lat: 51.388, lon: 30.099
}
```

Finally there are a few minor syntaxes, such as ellipsis and prefix dot which we will go over in detail later. And of course there are line comments introduced by a hash—which we have already seen.

In these examples, I have written eexprs suggestive of some underlying semantics so as to motivate the features. However, it is important to note that no specific semantics are imposed by eexprs. E.g. a client of eexprs could encode function calls with the Haskell-like `f x`, Lisp-like `(f x)`, Algol-like `f(x)`, Forth-like `x f`, or in innumerable other ways—assuming the client language supports function calls at all.

TODO: once I implement these features, document them here:

- mixfixes
- ascii <-> unicode

Abstract Syntax

TODO show the data type

FIXME: this paragraph is garbo. This grammar is representable by a simple ADT. To transform an eexpr term into a more specific syntax, you could simply pattern-match against this ADT. However, if you want to report (client) grammatical errors in context, then it it likely preferable to use an arrow-based api to match eexprs. I have implemented such such an api. It also includes an instance for `ArrowApp`, which technically means that it could be lifted into a monad; however, I am skeptical of the monadic style's ability to correctly track

pattern-matching context (without careful programmer diligence, which I know from personal experience to be a pipe dream).

Recognizer Combinators

Concrete Syntax

TODO this section should be a formal description of the grammar, along with informal explanations and rationales. Here should be the notes about how much work a parser implementation would need to do.

Lexer

Post-lexer

Grammar

Suboptimality of S-Exprs

TODO now that this is in a different part of the report, it needs new linking verbiage

As powerful as the recognizer technique is, I don't think s-exprs is its best medium. I don't want to rant about s-exprs too much, since that's not the point; if you stopped reading right now and used recognizers over s-exprs instead of writing parsers, your life would improve plenty. That said, if you have used enough s-exprs, you have probably experienced some pain points that e-exprs would improve on. At the other end of the spectrum, if you have resisted learning a Lisp because it looks too alien, then perhaps it would be better to jump directly to e-exprs.

The most important practical problem with s-exprs is a historical one: there is no standard for them. Different Lisp implementations have different notions of what constitutes an atom, and many have different concrete syntaxes for atoms even when they agree on the abstract syntax. Additionally, most implementations come with reader macros, which require a programming language to be well-defined before parsing into s-exprs can even be fully defined. Further, the "pure" s-exprs presented above have been "polluted" with a variety of syntactic shortcuts, such as the cons-dot `(a . b)`, a variety of quotation syntaxes, and syntax for keyword arguments; these extensions are another source of differences between implementations. The suspicious thing about these syntax extensions is that they are something no Lisp programmer would want to live without, but would contribute very little to non-Lisps.

The most common objection I've heard is that Lisps have "too many parenthesis". While this appears to be a surface complaint, it would be foolish to ignore such persistent "customer" feedback. A more cogent version of this complaint would be that the meaning of parenthesis in Lisp are grossly overloaded. When faced with new Lisp code, one must figure out what each set of parenthesis is used for: do they invoke macros? procedures? project a struct member? delimit a list? a mapping? the items in a list or mapping? code blocks? statements within a block? and so on… Given more syntax, a language designer has more opportunity to visually distinguish common programming constructs.

E-expr

```
match os.name:
  "linux": display "Hello, `fromMaybe name "Linus"`!"
  "mac": display "Hello, `fromMaybe name "Steve"`!"
  "windows": {
    display "Hello, `fromMaybe name "Bill"`!"
    init-wsl
  }
```

S-expr

```
(match (os-name os)
 ("linux" (display (string-concat "Hello, " (fromMaybe name "Linus") "!"))
 ("mac" (display (string-concat "Hello, " (fromMaybe name "Steve") ""))
  ("windows" (begin
   (display (string-concat "Hello, " (fromMaybe name "Bill") "!")
    (init-wsl))))
```

Of course, there are tools to help manipulate s-exprs, so one might argue that the ergonomics are not a real issue. However, tooling does not exist everywhere: unconfigured (or worse: badly-configured) editors, whiteboards and paper, web pages. A design philosophy I used for eexprs is "the easier it is to write with tooling, the easier it is everywhere"[3]. Indeed, while I'm developing eexprs, I have no tooling for them, but I find it no more difficult than using sexprs with tools.

Before moving on, I just want to point out one non-advantage of s-exprs. I have heard it argued that the minimalist syntax of Lisp is the price you pay for Lisp's most iconic advantage: its macro system. Indeed, one can very easily construct new sexprs from templates simply by introducing special forms for `quasiquote`, `unquote` and `unquote-splicing`. However, it is not necessary for s-exprs to be so minimalistic; it is sufficient that they form an algebraic data type. With an appropriate set of quasiquoters and unquoters, any language representable in an

[3]In a talk about accessibility, I picked up the idea that if you make something accessible for disabled people, you've probably made it easier for abled people as well. For example, a website that is usable by a person with dementia is not likely to be frustrating for the average user. This philosophy applies to the design of programming languages, and in a cybernetic sense, a lack of technology is a lack of ability.

ADT can also take advantage of quasiquotation to implement metaprogramming. Nevertheless, I can see why in the 1960's they went with sexprs rather than a richer system.[4]

NOTES

parse a "level 0/1" int by checking that `radix^exp` is less than `2^n` for some reasonable `n` (perhaps 64 for known small ints, but if you allow exact big ints, then some larger n like `8^m` for a max of `m` bytes of memory (maybe 4096 is good? just how big a bigint do you need?

If a colon has a newline on the right, it starts a block. If a colon has inline space on the right, it's a normal colon. If a colon has no space on the right, it might be part of a symbol: it joins up with any symbols to the right or left it also joins up with any colon to the right recursively That way, I can have `Foo:bar.baz` be the `baz` record of the `Foo:bar` qualified name, just as long as I also have a way to split strings on colons.

Because eexprs draw a line between two algorithms both commonly referred to as "parsing" a "grammar", we'll use

- "eexpr grammar" and "parsing" for the grammar of plain eexprs and the parser that grammar, respectively
- "client grammar" and "recognizing" for the grammar of a eexpr-based language and the parser/pattern matcher for that grammar, respectively

[4] Have you ever tried to implement a compiler in 4096 *bytes* of memory?