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.

A Whirlwind Tour of E-Exprs

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 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

[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.

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

At least conceptually, the grammar of eexprs can be split into four phases:

1. Decode UTF-8
2. Raw Lexer
3. Token Cooker
4. Parser

Raw lexing identifies individual tokens in a Unicode stream. It can be described mostly with regular expressions, though at points we require capturing groups. In the reference implementation, we require at most three (TODO verify) codepoints of lookahead.

Cooking the token stream involves examining tokens in context to determine if they are relevant to the parser, and sometimes disambiguate how the token is to be used (e.g. space that separates eexprs vs. space which indicates indentation). This is described by a term rewrite system which only uses a few (TODO how many?) tokens of lookaround.

Finally, the parser consumes the cooked token stream to produce an eexpr according to a context-free grammar.

The Unicode character set 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 flexible, but detection of magic strings in their appropriate encodings is a source of complexity. Would that complexity be worth it? I don't think so, and the UTF-8 Everywhere Manifesto presents the argument better than I can here.

Raw Lexer

TODO what all metasyntax do I use? only introduce it just before it is needed, not all at once

```
token ::= nl | lws(_, _) | lineJoin | comment
       | number | symbol
       | TODO
```

Whitespace TODO these are the bits of whitespace that are used all over the place

```
NL ::= <U+000A New Line>
CR ::= <U+000D Carriage Return>
RS ::= <U+001E Information Separator Two>
nl ::= NL CR? | CR NL? | RS
# additional types of newline may be considered
# it is also possible for implementations to ignore obsolete newline sequences; I've included
```

```
lws('SP', n ≥ 1) ::= <U+0020 Space)>{n}
lws('HT', n ≥ 1) ::= <U+0009 Character Tabulation>{n}
# additional types of space may be considered, but each must get a distinct whitespace type
```

TODO these are whitespace-like things

```
BS ::= <U+005C Reverse Solidus>
lineJoin ::= BS lws(_, _)? nl

comment ::= '#' (!nl)*
```

Numbers and Symbols E-exprs support both integer and fractional numbers in a variety of radices (bases). Doing this requires a surprising number of rules, but the logic is fairly intuitive.

At their core, numbers consist of an integer part, an optional mantissa, and an optional exponent. For numbers not in base ten, a different radix can be indicated by beginning the integer part with a radix-leader (`0x` and the like). When there is a mantissa, there must be digits on both sides of the decimal point (e.g. `0.0` is allowed, but neither `0.` nor `.0`). The exponent is essentially the same as the integer part, but only fractional numbers are allowed to have signed exponents. There are a variety of ways to indicate the start of an exponent, dependent on the base.

```
number ::= sign? intPart(base) uexp(base)?
        |  sign? intPart(base) fracPart(base) exp(base)?

intPart(base) ::= radixLeader(base) digits(base)
fracPart(base) ::= '.' digits(base)

uexp(base) ::= expLetter(base) digits(base)
            |  '^' radixLeader(expBase) digits(expBase)
exp(base) ::= sign? uexp(base)
```

Fractional numbers must have both integer and mantissa present, since there's really no reason to omit either. I think the extra character on either side is a negligible cost; if you find it it significant, I would wonder why you have so many magic constants in your code. Plenty of style guides require the zero in all or most contexts, and including it makes space in the grammar to distinguish numbers from other syntaxes. While the prefix-dot syntax I discuss below would have conflicted with an optional zero integral part for numbers, I had already decided on the numerical syntax before developing the prefix-dot. Perhaps it was dumb luck that I didn't have to work hard to squeeze an idea into the syntax, but I flatter myself to think that early preparation enabled that luck.

E-exprs support bases 2, 8, 10, 12, and 16. Unsurprisingly, base 10 is the default, and the digits are exactly as you might expect.. To allow visual grouping, digits may be separated by underscores, but underscores should not appear at the

start of end of a string of digits.

```
digits(base) ::= digit(base)+ (('_' | digit(base))* digit(base))?

digit(2) ::= '0' | '1'
digit(8) ::= '0'…'7'
digit(10) ::= '0'…'9'
digit(12) ::= radixChar(10) | ('␘' | 'X') | ('␗' | 'E')
digit(16) ::= radixChar(10) | 'a'…'f' | 'A'…'F'
```

Allowing underscores in numbers makes it possible to group digits in ways that make large numbers easy to identify by subitizing rather than the more costly method of counting. Allowing multiple underscores makes it possible to have multiple levels of grouping; as numbers get larger, it becomes difficult to idenify how many groups there are. Consider `18_446_744__073_709_551_616` vs. `18_446_744_073_467_709_551_616` vs. `18446744073709551616`: it is easy to see that the first is about 18 million billion, but in the second I have to count the groups to estimate 18 *billion* billion, and in the third I have to count every digits to verify that it's back to 18 million billion. I picked this example specifically because I had trouble remembering roughly how much 2^{64} is: unlike 2^{32}, there are just too many thousands groups to subitize.

Base twelve is thrown in there basically just for fun, though it does help that the Unicode Consortium added dedicated codepoints for dec and el digits. The alternate ASCII glyphs `X` and `E` were chosen because the Dozenal Society of America voted for those forms where Unicode is poorly supported.

More seriously, I had considered bases 32, 62, and 64. However, the alphabets for these bases are not widely agreed-on, so I leave those notations to client grammars.

If it were easier to input Cuneiform, and I had a good way to solve the problem of representing zeros, I might have added *it* too—again for the lulz.

Radix indicators are the commonly-used ones. There is no indicator for ten (and I have marked this explicitly in the grammar), since base ten is the default. Exponents—you may have seen above—can always be marked with a `^` character, which also allows a change of radix.[3] There are also some radix-specific exponent markers for radices where there is some agreement: namely in bases 2, 10, and 16.

```
sign ::= '+' | '-'

radixLeader(base) ::= '0' radixLetter(base)
radixLeader(10) ::= ε

radixLetter(2) ::= 'b' | 'B'
radixLetter(8) ::= 'o' | 'O'
```

[3]Why you would want to change bases between base and exponent I don't know…

```
radixLetter(10) ::= Ø
radixLetter(12) ::= 'z' | 'Z'
radixLetter(16) ::= 'x' | 'X'

expLetter(2) ::= 'b' | 'B'
expLetter(8) ::= Ø
expLetter(10) ::= 'e' | 'E'
expLetter(12) ::= Ø
expLetter(16) ::= 'h' | 'H'
```

Eexpr implementations should be careful when choosing an internal representation for parsed numbers. It is possible for a naïve implementation to consume all system memory. For example, numbers such as $10^{(10^{100})}$ can be represented in an eexpr with as few as 103 bytes, but when represented as an exact bigint, would require more memory than humans have ever manufactured. In the reference implementation, we do not attempt to expand the exponent, and instead leave it to client languages to determine whether (and how) to represent a number or emit an error. To help with this (TODO implement this), we have also included grammar combinators which only succeed for numbers that fit into various common representations.

Strings Eexprs support three string formats to support different levels of escaping within the string. They are:

- C-style (or double-quoted strings),
- SQL-style (or single-quoted strings), and
- heredocs (or triple-quoted strings[4]).

```
string(x, y) ::= cString(x, y)
string(S, S) ::= sqlString
              |  heredoc
```

C-style strings have the richest syntax for escapes, and are the only string style that can be used as a template. Instead of attempting to detect nesting level in the lexer, we instead detect fragments of templates and leave it to the parser to attempt assembly into sensible string template expressions. We therefore categorize C-style strings by how they might join with other string parts.

```
SQ ::= <U+0027 APOSTROPHE>
DQ ::= <U+0022 QUOTATION MARK>
BQ ::= <U+0060 GRAVE ACCENT>

cStrDelim(S) ::= DQ
cStrDelim(T) ::= BQ
cStr(x, y) ::= cStrDelim(x) cStrUnit* cStrDelim(y)
```

[4] even three double-quotes would be six quotes…?

```
cStrUnit ::= cStrChar
           |  BS cEscapeChar
           |  BS cStrLineJoin BS

HEX ::= '0'…'9' | 'a'…'f' | 'A'…'F'
cEscapeChar ::= BS | SQ | DQ | BQ
              |  'n' | 'r' | 't'
              |  '0' | 'e' | 'a' | 'b' | 'f' | 'v'
              |  'x' HEX{2} | 'u' HEX{4} | 'U' HEX{6}
              |  '&'
cStrLineJoin ::= nl lws(_)

# TODO I should likely eliminate non-printing characters as well
cStrChar ::= 1 - (BS | DQ | BQ | nl)
```

While most of the escape sequences are taken from C, a few are less-widely represented. The sequence `\e` encodes `<U+001B ESCAPE>`, and the sequence `\&` encodes a zero-length string. The `\x`, `\u`, and `\U` sequences encode arbitrary codepoints from (respectively) ASCII, the Unicode Basic Multilingual Plane, and all of Unicode. Attempting to encode a codepoint outside the range of Unicode (anything larger than 0x10FFFF) is an error.

To be honest, I'm not sure that I need both `\&` and the fixed-length `\x`/`\u`/`\U` escapes. The former was stolen from Haskell which allows variable-length Unicode escapes (e.g. `"\x333" !=“33”== \x33\&3`). The latter is my own solution (though apparently C has something similar) to deal with the potential ambiguity of variable-length Unicode escapes by eliminating variable length entirely. It is entirely possible that I may change this area of the syntax before a 1.0 release in response to feedback.

While `\e` is not an escape compliant with the C Standard, this was because some character sets (e.g. EBCDIC) lack a character with the corresponding ASCII semantics. In the case of eexprs—which are explicitly reliant on Unicode—this is not a concern. Meanwhile, `\e` is used sometimes for terminal programming, and would likely be more meaningful for custom binary formats than the obscute “vertical tab” `\v` sequence.

Two C escapes `\?` and octal `\nnn` are missing from eexprs. The `\?` escape is to avoid trigraph syntax, a rarely-used C feature which eexprs do not include. Octal escapes on the other hand, are perfectly reasonable, but it seems few programmers use octal today. Even DEC, which famously used octal in all their PDP-11 and VAX documentation, switched to hexadecimal notation when they published the Alpha architecture.

SQL-style strings offer fewer escapes, but a compensatorily wider range of valid characters. SQL-style strings are delimited by single quotes, but a sequence of two single-quotes encodes a single quote into the string.

```
sqlStr ::= SQ sqlStrUnit* SQ
```

```
sqlStrUnit ::= 1 - SQ
             |  SQ SQ
```

I had considered using single-quotes to indicate character literals. However, I was also envious of syntax such as Python's r-strings for easily writing strings which have lots of backslashes, such as regex. Instead of hardcoding a fixed (and therefore small) set of string prefixes, I instead opted to allow dot-separated symbol+string to omit the dot. Client languages can then determine any special interpretations for such a string, perhaps interpreting a string as a character. This then made space for SQL-style strings, which like Python r-strings can encode C-style escapes with ease.

In typed languages, Haskell's `IsString` should serve as an inspiration: interpret a string literal as a character literal in contexts where a character-typed value is expected. The same technique can also be used for alternate memory layouts of strings, hexadecimal strings, base64 strings, ip addresses, mac addresses, and so on for as many types as users wish they had literals for.

In untyped languages, conversion functions (in general) cannot be inferred, and so must be manually inserted. This could be done by prefixing a string with a conversion function (perhaps in a separate namespace). Thus, `c'a'` for a character, `base64url"KMK044O7z4njg7sp44Gj55Sx"` for a base64-encoded bytestring, and so on.

TODO heredocs

TODO heredocs are here mostly for docstrings. while many languages put documentation in comments, I like Python's ability to print documentation for an object from the repl. putting such documentation in comments would make it difficult to attach it to language objects, since comments are meant to be removed before parsing.

the other use they might have is in embedding other languages, such as sql.

Punctuation TODO

```
punctuation ::= encloser(_, _)
              | separator(_)
              | ellipsis
              | unknownDot
              | unknownColon

encloser(Open, Paren) ::= '('
encloser(Open, Bracket) ::= '['
encloser(Open, Brace) ::= '{'
encloser(Close, Paren) ::= ')'
encloser(Close, Bracket) ::= ']'
encloser(Close, Brace) ::= '}'
```

```
separator(Semicolon) ::= ';'
separator(Comma) ::= ','

unknownDot ::= '.'
unknownColon ::= ':'
ellipsis ::= '..' # TODO probly also include '…'
```

Lexeme Cooker

The lexeme cooker is a sequence of rewrite rules. TODO: I am less familiar with the properties of rewrite systems, but I think these could be applied each in-order in a single pass over the string, or make several passes one for each. TODO: I'm also not sure how much these rules could be re-ordered, but I do know some must come before others (e.g. whitespace classification before indentation). Rules listed earlier in this system take precedence over rules listed later.

Before applying this system, the token stream is augmented with a start-of-file token `SOF` at the start and an `EOF` token at the end. Implementations perhaps need not actually allocate space for such tokens, but this practice at least makes it easier to specify.

TODO A few rules don't really fit anywhere. In the reference implementation, it is a warning for a file to use two different types of newline. Trailing whitespace is a warning.

`MIXED_SPACE, MIXED_NEWLINES, MIXED_INDENTATION, TRAILING_SPACE, NO_TRAILING_NEWLINE`

It is recommended for implementations to emit warnings in these cases, and allow users to suppress them or escalate them into errors.

`!sol EOF ⇒ Ø?`

Given a set of valid files of eexprs that end with at least one trailing newline, any simple concat of these files would produce a valid file of eexprs. This property could come in handy when assembling eexprs mechanically. In particular, a compiler could simply prepend the dependencies of a module and compile the whole program at once, rather than needing to define an interface file format and perform linking. One of the goals of eexprs is to make it faster to produce reasonable quality experimental languages, so a feature that makes it easier to support multi-file source code is a no-brainer.

In the endless religious war of tabs vs. spaces, there is only one real issue that bugs me: alignment. Namely, I think humans should not be aligning code by hand—it's too fragile to justify the tedium. Somehow, we need to get machines to align code for us. Proposals have been made to enhance editors with smarter features, but I think it is unreasonable to expect all editors to choose the same methodology, or even to upgrade at all.

I propose a separate tool which can operate over any text file, searching for special characters that indicate where alignment into a table should take place. Unfortunately, there isn't a clearly good choice for an alignment indicator character. Importantly, whatever indicator is chosen should remain in-place so that code can be re-indented without re-inserting the indicator; thus, we cannot simply use any sequences of tabs and spaces, even if we were to ban one of them from all other uses. If we could just pick a codepoint without care, I would probably go for a zero-width space, or possibly repurpose one of the disused ASCII control characters; these are unfortunately not easy to type on any standard keyboard. For accessibility, backslash-space might be the best, but I expect it to have poor aesthetics, and possibly have complex interactions with tabs characters.

In any case, it is reasonable to expect that some whitespace handling in eexprs will be adjusted in the future to enable the use of an external alignment tool.

Managing Whitespace

```
sol ::= nl | SOF
eol ::= nl | EOF
```

TODO mixed whitespace is an error

```
lws(x, _) lws(y ≠ x, _) ⇒ ∅
```

It should be impossible for the lexer to produce two `lws` tokens of the same type, but just in case, the rule `lws(x, m) lws(x, n) ⇒ lws(x, n + m)` would fix up such an issue.

TODO ignore comments, trailing newline, line continues, and blank lines

```
lws(_)? comment? eol ⇒ eol
lws(_) lineJoin ⇒ lineJoin
lineJoin lws(n) ⇒ lws(n)
nl eol ⇒ eol
```

TODO colons at the end of a line indicate indentation, otherwise are separating colons FIXME and also I should disallow separating colons without a space afterwards TODO and also a space is inserted when needed

```
unknownColon nl lws(n) ⇒ indent(n)
unknownColon nl ⇒ indent(0)
unknownColon EOF ⇒ indent(0) EOF
unknownColon ⇒ colon

(?before ≠ space | sol | unknownDot) indent ⇒ \before space indent
```

TODO indentation is weird 'cause I need a stack. I mean, I could thread an attributeful token through the stream, but I think the stack explanation is

clearer. Perhaps I annotate the rewriter with a state when I need to `A (i = [3, rest..])⇒(i = [rest..]) B`.

TODO at this point, that all nl and lws have been rewritten

```
nl | lws(_) ⇒ ∅
```

TODO spaces are irrelevant inside enclosers

```
encloser(Open, a) space ⇒ encloser(Open, a)
space encloser(Close, a) ⇒ encloser(Close, a)

cStr(a, T) space ⇒ cStr(a, T)
space cStr(T, a) ⇒ cStr(T, a)
```

TODO dots with no space around are chain dots. TODO dots with space only before are prefix dots TODO other dots are errors

```
(?1 ≠ space | sol) unknownDot (?2 ≠ space | eol) ⇒ \1 dot \2
(?1 = space | sol) unknownDot (?2 ≠ space | eol) ⇒ \1 prefixDot \2
unknownDot ⇒ ∅
```

TODO punctuation built out of dots cannot be adjacent TODO dot after a number is an error (confusable with a trailing decimal point TODO symbol and number cannot be adjacent (confusable as a single symbol) TODO string next to string is an error (looks too much like implicit catenation?) detectCramming(st);

```
(dot | ellipsis | prefixDot) (ellipsis | dot) ⇒ ∅
number dot ⇒ ∅
(number | symbol) (number | symbol) ⇒ ∅
cStr(_, S) cStr(S, _) ⇒ ∅
```

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"[5]. Indeed, while I'm developing eexprs, I

[5]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.

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 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.[6]

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

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