Report on E-Exprs

Eric Demko

This report describes a grammar and parser for a data language with optimized ergonomics for encoding a wide range of programming languages. They key technical 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 main contribution is a selection of syntax that is familiar and therefore intuitive for programmers to read and write. The rest of this report will define eexprs, explicate the implementation technique, and motivate/rationalize my design choices.

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.

```
# 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 to which separators can be used with which enclosers, or even that either needs

[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.

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. Commas are optional both before and after a comma-separated expression, and likewise for semicolons. With proper supoprt from the client language, we can use indented blocks to allow the omission of oft-forgotten end-of-line commas.

```
{
  type: "point"
  altitude: 0
  , 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

E-exprs form a recursive algebraic data type, where each constructor represents a widely-recognized programming construct:

```
data Eexpr
  = Symbol    ShortText
  | Number    Bignum
  | String    ShortText [(Eexpr, ShortText)]
  | Paren     (Maybe (Eexpr))
  | Bracket   (Maybe (Eexpr))
  | Brace     (Maybe (Eexpr))
  | Block     (NonEmpty (Eexpr))
```

```
  | Predot    (Eexpr)
  | Chain     (NonEmpty2 (Eexpr))
  | Space     (NonEmpty2 (Eexpr))
  | Ellipsis  (Maybe (Eexpr)) (Maybe (Eexpr))
  | Colon     (Eexpr) (Eexpr)
  | Comma     [Eexpr]
  | Semicolon [Eexpr]

data NonEmpty a = a :| [a]
data NonEmpty2 a = a :|| NonEmpty a
```

A sequence of dot-separated expressions is represented with the `Chain` constructor. I thought the name `Dots` was a little misleading because the dots are sometimes optional, as in `array[i]`, which is a `Chain` that contains no dots at all in the concrete syntax.

The one constructor that isn't widely-recognized is `Predot`, which represents an expression prefixed by a single dot.

> *Rationale*
>
> In Haskell, named functions can be made into binary infix operators—e.g. `x` `` `member` `` `s` rather than `member x s`. Originally I wanted to generalize this so that arbitrary expressions can be made into infix operators—e.g. `map .(insert k) v`. However, I quickly realized that this could also be used to create "artifical methods"—e.g. rewrite `get theMap .insert k (compute blarg)` to `insert (get theMap) k (compute blarg)`. Even more generally, a prefix dot could be used to determine an alternate meaning of the expression it attaches to. In pattern matching for example, one must determine if a symbol is a constructor or a named hole; a prefix dot could be used on holes to distinguish them, rather than (say) enforcing a case convention. Regardless, it is up to the client language to determine the meaning of a prefix dot in various contexts, if it allows prefix dot at all; eexprs simply make the syntax available if there *is* a need.

Recognizer Combinators

FIXME: this paragraph is garbo. 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).

TODO ArrowApply is implemented so you can do things like (roughly)

```
specialForms :: Map Text ([Eexpr] ~> Lang)
```

```
parseSpecialForm :: NonEmpty Eexpr ~> Lang
parseSpecialForm = proc (x :| xs) -> do
  isSpecialForm <- lookup x specialForms
  case isSpecialForm of
    Nothing -> pure $ Apply x xs
    Just form -> form xs
```

Perhaps I don't know arrows well and haven't figured out how to write this without `app`, but it's odd that if I inline the table as a syntactic case instead of a map datum, I can write:

```
parseSpecialForm :: NonEmpty Eexpr ~> Lang
parseSpecialForm = proc (x :| xs) -> case x of
  "lambda" -> parseLambda xs
  …
  _ -> pure $ Apply x xs
```

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

Rationale

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 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 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.

I've presented the parts of this pipeline in "reverse" order. AFter all, the rest of this paper is primarily a long list of definitions of tokens and terminals and

character sets and… I hope you'll be less distracted by questions like "Why are you defining this?" if you first see where it will be used. Of course, if you prefer to trudge through several dozen low-level definitions and slowly build up to anything useful, you're welcome to read backwards. But for now, the parser:

Parser

The true point of the raw lexer is to handle tricky parts of decoding a character stream, and the lexeme cooker handles almost all the tricky work of making sure the tokens relate to each other intuitively, so the parser is refreshingly straightforward. There are some final intuition-driven intricacies for parsing the lines of indented blocks, but we will get to that later.

The core of eexprs is the `Expr` rule; this is the start rule and is called back into to tie the recursive knot. It can be composed of an eexpr delimited on both sides (bracketed eexprs, indented blocks, and string templates), or an atom (which, since it is always one token long, can be said to delimit itself).

```
Expr ::= number
       | symbol
       | string(S, S)
       | EnclosedExpr

EnclosedExpr
  ::= encloser(Open, a) SemicolonExpr? encloser(Close, a)
   | indent(n) BlockLine (nl BlockLine)* dedent(n)
   | string(S, T) SpaceExpr (string(T, T) SpaceExpr)* string(T, S)
```

Rationale

TODO why string templates hold spaces? I guess because you're supposed to put a single (client lang) expression inside to be spliced in; what would it mean to put statements inside? If someone does come up with a use-case, I can extend the grammar later.

Beyond that, eexprs may be combined by various punctuation. It seems odd to speak about punctuation having precedence, so we will say that each form of punctuation has a different "strength", with stronger punctuation appearing closer to the root of the parse tree. From strongest to weakest we have: semicolon, comma, colon, ellipsis, space, dot. Note that although bracketed expressions contain up to `SemicolonExpr`s, string templates contain `SpaceExpr`s (and again, indented blocks are a little odd).

```
SemicolonExpr ::= semicolon? CommaExpr (semicolon CommaExpr)* semicolon?
                | semicolon

CommaExpr ::= comma? ColonExpr (comma ColonExpr)* comma?
            | comma
```

```
ColonExpr ::= EllipsisExpr (colon EllipsisExpr)?

EllipsisExpr ::= SpaceExpr? ellipsis SpaceExpr?
               |  SpaceExpr

SpaceExpr ::= ChainExpr (space ChainExpr)*

ChainExpr ::= Expr ChainTail*
ChainTail ::= dot Expr
            |  EnclosedExpr
```

Each type of punctuated expression allows different numbers of child expressions:

- Semicolons and commas both allow zero or more child expressions. When there are two or more, everything is straightforward; though the leading and trailing commas are both optional. To semicolon/comma separate a single expression, at least one of the leading/trailing commas must be present. A single semicolon/comma with no subexpressions on either side encodes zero separated eexprs.
- Colons require children on both sides.
- Ellipses allow a child on both sides, but do not require either; which child is on which side is meaningful.
- Spaces and dots allow two or more child expressions.

In each line of a block, colons temporarily become sort-of stronger than semicolons. Intuitively, a block line holds semicolon-separated eexprs, but is also allowed a left-hand side before a colon, which is at most an `EllipsisExpr`. This easy-to-describe change requires the grammar for `BlockLine` to be rather more involved, unfortunately, but I guess that's the nature of squishy human think-meat. To simplify the presentation, we use some explicit lookahead in these rules, which is notated with angle brackets.

```
BlockLine
  ::= <semicolon | comma> SemicolonExpr
   |  EllipsisExpr <semicolon> SemicolonExpr
   |  EllipsisExpr <comma> CommaExpr (<semicolon> SemicolonExpr)?
   |  EllipsisExpr (colon SemicolonExpr)?
```

Rationale

I claim that the "left-hand side" of block lines is intuitive, but at first block lines were simple semicolon-separated expressions, and why shouldn't they be? Well, despite designing and implementing eexprs, I immediately screwed up writing a recognizer for mixfixes. I imagined that

```
mixfix add:
  after: mul, exp
```

would parse the obviously-intended way, but instead it grouped `after: mul`

together, leaving `exp` as a sibling. It's certainly interesting that sometimes the intuitive rule leads to unintuitive results. One small change to the grammar later, and my recognizer was far easier to write. In any case, I realized that I wanted a LHS rule anyway to support switch statements (and case/match expressions, and so on), such as:

```
switch c:
  'A'…'F': c -= 7; fallthrough
  '0'…'9': n = n * 10 + (c - 48)
  default: return null
```

Lexeme Cooker

The lexeme cooker is a string rewrite system over tokens. It deals with mostly local interactions between tokens, especially normalizing whitespace and whitespace-sensitive tokens.

The basic form of rewrite rule is `a ⇒ b` which replaces the token sequence `a` in the input with tokens `b` for output. We also use lookahead and lookbehind (collectively lookaround) in the left-hand side of a pattern. These affect only whether the rewrite rule fires, but the sequences matching the lookaround are not rewritten. We write `<l>` for lookaround which must match for a rule to fire, and `!<l>` for lookaround which must *not* match; of course, lookaround is only allowed at the start and/or end of the left-hand side of a rule. Within lookaround sequences, we also use some "non-terminal" patterns; the patterns we use can be modeled as regular languages over tokens, so they're really just convenient shorthands.

The rewrite system may also be required to report an error, which is written `lhs ⇒ ERR`. There are a number of places where an implementation may warn the user, written `lhs ⇒ WARN rhs`, or as `lhs ⇒ WARN` when no rewriting takes place. With appropriate configuration, warnings may also be elevated to errors, but eexprs which generate warnings are nevertheless valid and should be accepted by default.

Rationale

The characteristic that distinguishes some conditions as warnings rather than errors is that those conditions could be unambiguously machine-corrected in the source code without a change to the resulting eexpr.

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 cooking, the raw token stream is augmented with a start-of-file token `SOF` at the start and an `EOF` token at the end. Once cooking is complete, these

special tokens are then removed. Implementations perhaps need not actually allocate space for such tokens; this practice just makes it easier to specify.

Oddball Rules Two commonly-used token patterns are for start- and end-of-line.

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

It is recommended that every file of eexprs end with a trailing newline.

```
!sol EOF ⇒ WARN
```

Rationale

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.

Mixed whitespace (adjacent tabs and spaces) is an error. It should be impossible for the lexer to produce adjacent whitespace tokens of the same type, but just in case, we can also merge such tokens at this point.

```
lws(x, _) lws(y ≠ x, _) ⇒ ERR
lws(x, m) lws(x, n) ⇒ lws(x, n + m)`
```

Some rules are cumbersome to specify in this rewrite system, but are simple enough that prose should suffice:

- If there are two types of indentation (start-of-line tabs-vs-spaces) in-use, that is an error (TODO iirc, this includes heredoc indentation, not just line-leading whitespace tokens).
- If a single file uses two or more types of newline, that is a warning.
- Whenever there is trailing whitespace token (not necessarily whitespace characters, which may be part of a token), that is a warning.

Rationale

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.

Normalize Whitespace Comments and blank lines are ignored, as is intuitive. When joining lines, only the leading whitespace of the final line is retained.

```
lws(_, _)? comment? <eol> ⇒ ε

lws(_, _) lineJoin ⇒ lineJoin
lineJoin lineJoin ⇒ lineJoin
lineJoin lws(_, n) ⇒ lws(_, n)

nl <eol> ⇒ ε
```

Linear whitespace immediately inside enclosers is ignored. E.g. `(a)` and `( a )` are indistinguishable to the grammar. The same logic applied to string templates, with appropriate adjustments for the middle parts of the string.

```
<encloser(Open, _)> lws(_, _) ⇒ ε
lws(_, _) <encloser(Close, _)> ⇒ ε

<cStr(_, T)> lws(_, _) ⇒ ε
lws(_, _) <cStr(T, _)> ⇒ ε
```

Indentation The logic for indentation is that an end-of-line colon indicates that the next leading whitespace defines an indent level. Each logical/abstract line of an indented block must begin with the same amount of leading whitespace that defines the level of the block. A logical line may span multiple physical lines, but only of it obeys the "offsides rule": extra physical lines must begin with more leading whitespace than the level of indent. Dedents are inserted whenever a line begins with less than the current indent level. However, dedents must return to an active indentation level.

```
block1:
  block2: # activate an indent at two spaces
    do lots of stuff # activate an indent at four spaces
    do lots    # | these two lines are equivalent
      of stuff # | to the line above
```

```
  end block2 # deactivate the four-space indent
 illegal # | this line starts with one space,
         # | which is not an active indentation level
```

It is easy to insert indentation, since it is always starts at an end-of-line colon or open encloser. We also insert a synthetic space before indentation-colons, unless that colon is preceded by a dot. At the same time, we can classify other unknown colons. Separator colons must be followed by linear whitespace, but space around them is irrelevant to the grammar. Colons not followed by whitespace (linear or newline) are an error.

```
!<sol | unknownDot | lws(_, _)> unknownColon nl lws(_, n) ⇒ space indent(n)
!<sol | unknownDot | lws(_, _)> unknownColon !<lws(_, _)> ⇒ space indent(0)
<sol | unknownDot | lws(_, _)> unknownColon nl lws(_, n) ⇒ indent(n)
<sol | unknownDot | lws(_, _)> unknownColon !<lws(_, _)> ⇒ indent(0)
<encloser(Open, x)> nl lws(_, n) ⇒ indent(n)
<encloser(Open, x)> ε !<lws(_, _)> ⇒ indent(0)

lws(_, _)? unknownColon lws(_, _) ⇒ colon
unknownColon !<lws(_, _)> ⇒ ERR # FIXME implement this in C
```

Rationale

I expect that indented blocks will much more often be the final element of a space-separated sequence of eexprs, rather than the final element of a dot-separated sequence. Thus, rather than group indent/dendent with other enclosers, if you wish to dot-separate an indented block, the dot is necessary. Inserting a synthetic space makes lines such as `do:` and `do :` equivalent, and thus allows the indentation colon to be placed directly against an eexpr rather than requiring an unnatural space to separate them. If a block at the end of a dot-separated sequence is desired, explicit dots `do.:` can achieve it.

Rationale

When an open encloser is at the end of the line, the following lines are almost always indented, be it for statements in a block, elements in a list, arguments for a function, or what-have-you. Thus, eexprs automatically begin an indent immediately after a line ending with an open encloser. This may seem redundant, but is exactly this redundancy which helps detect inconsistencies between the structure of bracketing and the structure of layout, which have caused security vulnerabilities (at least one I know of, which means there must be others that I do not know of, and others still that were not made public). This behavior can be suppressed if desired with line joining.

Rationale

I'm still debating whether to allow colons in symbols, and whether to allow fat colons. By requiring space after a colon, any decision can be made backwards-compatibly.

FIXME I'm thinking more about whether to join a colon-nospace-symbol into a single symbol, but there's not a good way to express this in the rewrite rules.

Perhaps it can go in the symbol raw lexer rule for symbols, but it doesn't belong to this phase. I guess for now I'll require lws after regular colons, just to play it safe. It's not just '::' that I want, but also ':=' and especially '::='. It seems I really do want to allow colons in identifiers in some capacity; I just have to make sure trailing colons are disallowed.

You may be able to see that the logic for inserting dedents is not presentable with the rewrite system we have so far. Instead, we will need to augment it with a stack. Now that we need it, we introduce the notation `a s⇒t b` for a rewrite rule which updates the initial state `s` to `t` while also rewriting `a` to `b`. When `s` is left out, that indicates that the initial state is irrelevant, and when `t` is left out, there is no change to the state. We use a non-empty cons list to as the stack, and use Haskell-like notation for it (`x:xs` for cons, and `[x]` for singletons).

These next rules use the state to ensure that indents are valid (they are deeper than any active indent) and insert dedents. The initial state for the rewrite system is `[0]`. It makes sure that dedents are valid by using two rules for when leading whitespace is greater than the active indentation level. If we are not already dedenting, we begin dedenting, which leaves a trail of dedent tokens behind. So, if we detect leading whitespace greater than the active indent level, but have just seen a dedent, we know that a dedent has failed to return to a known indent.

```
indent(n > lvl) (lvl:rest)⇒(n:lvl:rest) indent(n)
indent(n ≤ lvl) (lvl:rest)⇒ ERR

nl zlws(n < lvl) (lvl:rest)⇒(rest) dedent(lvl) nl lws(n)
nl zlws(n = lvl) (lvl:rest)⇒ nl
!<dedent(_)> nl zlws(n > lvl) (lvl:rest)⇒ space
<EOF> (lvl:rest)⇒(rest) dedent(lvl)

<dedent(_)> nl zlws(n > lvl) (lvl:rest)⇒ ERR
```

Rationale

I experimented with no requiring dedents to return to a known indentation level, which would be a simpler algorithm to describe. However, I think it is too difficult to see when there are spaces after a dedent, even though that makes a significant change to the structure of the eexpr. Client languages which wish to space-separate blocks should instead enclose those blocks. For example, the following is a valid eexpr with two space-separated (enclosed and) indented blocks

```
if condition then {
  thing 1
  thing 2
} else {
  other thing
}
```

Out final rule for indentation is that the first eexpr in a file cannot have leading

whitespace. Then, all other eexprs in a file are (proper) subexpressions of the first, or else part of another eexpr on a different logical line.

```
SOF lws(n) ⇒ ERR
```

At this point, all trailing whitespace has been removed and all leading whitespace converted into indentation structure. Thus, we can transform all remaining linear whitespace into simple space tokens.

```
lws(_, _) ⇒ space
```

Space-Sensitive Constructs Finally, we look at tokens which are sensitive to surrounding whitespace. The first of these are dots, which must be classified into

- prefix dots, which have a space before them, but not after,
- chain dots, which have no space before or after, and
- all other combinations of dot and whitespace are illegal.

```
!<space | sol> unknownDot !<space | eol> ⇒ dot
<space | sol> unknownDot !<space | eol> ⇒ prefixDot
unknownDot <space | eol> ⇒ ERR
```

Finally, we are left with token sequences which can be difficult for humans to read quickly. For example, `1+2` would parse as a sequence of two tokens `1` and `+2`, and `a+b` would parse as a single symbol. We rule these sequences out with the following rules:

```
# symbols and numbers have too much in common to smush together
(number | symbol) (number | symbol) ⇒ ERR

# don't make readers try to count dots in a row
(dot | ellipsis | prefixDot) (ellipsis | dot) ⇒ ERR

# a dot after a number could be confused for a decimal point
number dot ⇒ ERR

# adjacent strings looks too much like C implicit string catenation
cStr(_, S) cStr(S, _) ⇒ ERR
```

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 e

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)
```

Rationale

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) | ('2' | 'X') | ('3' | 'E')
digit(16) ::= radixChar(10) | 'a'…'f' | 'A'…'F'
```

Rationale

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 digit 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.

Rationale

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 `^` char-

acter, 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'
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.

Symbols TODO TODO make sure to alter symbolChar later if I change the name here FIXME implement these choices in C

```
symbol ::= symbolChar1 symbolChar* - sign digit(10) symbolChar*

symbolChar ::= alphaNum | asciiSymb
             |  letterlikeSymb
             |  greekAlpha | mathHebrew
             |  blackboard | mathAlpha
             |  superscriptChars | subscriptChars
             |  mathSymb | arrows
             |  unlikelyMath | unlikelyArrows
             |  unsortedCharsTODO
```

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

```
symbolChar1 ::= symbolChar - SQ


alphaNum ::= 'a'…'z' | 'A'…'Z' | '0'…'9' | '2' | '3'
asciiSymb ::=
  | '!' | '$' | '%' | '&' | SQ | '*' | '+' | '-' | '/'
  | '<' | '=' | '>' | '?' | '@'
  | '^' | '_'
  | '|' | '~'

letterlikeSymb ::=
  | 'ℓ' | '№' | '℘' | 'ℏ'
  | '?' | '†' | '‡' # suggested for "unsafe" functions
greekAlpha ::=
  | 'A'…'Ω' - <U+03A2> | 'α'…'ω'
  # variant forms
  | 'ϐ' | 'ϴ' | 'ϑ' | 'ϒ' | 'φ' | 'ϖ' | 'ϰ' | 'ϱ' | 'ϵ'
mathHebrew ::=
 # note that these are not part of the unicode hebrew script; they are from the
  # Letterlike Symbols block, so they are typeset left-to-right
  | 'ℵ'…'ℸ'

blackboard ::=
   '𝔸'…'𝕫' - <reserved points in this range> | 'ℂ' | 'ℍ' | 'ℕ' | 'ℙ' | 'ℚ' | 'ℝ' | 'ℤ'
  | '𝟘'…'𝟡'
  | '𝕀' | '𝕆' | '𝕋' | '𝕌' | '𝕐' # greek
mathAlpha ::= # alternate alphanum fonts for math (𝒜𝒶, 𝔄𝔞)
 # TODO I removed bold fraktur b/c it's not different enough from normal fraktur
    '𝒜'…'𝓏' - <reserved points in this range>
  | 'ℬ' | 'ℰ' | 'ℱ' | 'ℋ' | 'ℐ' | 'ℒ' | 'ℳ' | 'ℛ'
  | '𝒾' | '𝒹' | '𝑜'
  | '𝔄'…'𝔷' - <reserved points in this range>
  | 'ℭ' | 'ℌ' | 'ℑ' | 'ℜ' | 'ℨ'

# as Unicode adds more miniscule latin sub/superscripts, these should be accepted as well
superscriptChars ::=
   'ᵃ' | 'ᵇ' | 'ᶜ' | 'ᵈ' | 'ᵉ' | 'ᶠ' | 'ᵍ' | 'ʰ' | 'ⁱ' | 'ʲ' | 'ᵏ' | 'ˡ' | 'ᵐ'
 | 'ⁿ' | 'ᵒ' | 'ᵖ'     | 'ʳ' | 'ˢ' | 'ᵗ' | 'ᵘ' | 'ᵛ' | 'ʷ' | 'ˣ' | 'ʸ' | 'ᶻ'
  | '⁰' | '¹' | '²' | '³' | '⁴' | '⁵' | '⁶' | '⁷' | '⁸' | '⁹'
subscriptChars ::=
   'ₐ'              | 'ₑ'          | 'ₕ' | 'ᵢ' | 'ⱼ' | 'ₖ' | 'ₗ' | 'ₘ'
 | 'ₙ' | 'ₒ' | 'ₚ'       | 'ᵣ' | 'ₛ' | 'ₜ' | 'ᵤ' | 'ᵥ'        | 'ₓ'
  | '₀' | '₁' | '₂' | '₃' | '₄' | '₅' | '₆' | '₇' | '₈' | '₉'

mathSymb ::= setSymb | logicSymb
          |  arithSymb | calculusSymb
```

```
          |  mathOps | mathRels | mathTurnstiles
          |  mathShapes
          |  ''' # prime
          |  nArySymb

setSymb ::=
    'Ø' | '∞'
  | '∩' | '∪' | '⊔' | '×'
  | '∈' | '∋' | '∉' | '∌'
  | '⊂' | '⊃' | '⊆' | '⊇' | '⊄' | '⊅' | '⊈' | '⊉'
logicSymb ::=
    '⊤' | '⊥'
  | '∀' | '∃' | '∄'
  | '¬' | '∧' | '∨' | '⊻' | '⊼' | '⊽'
  | '⅋' # linear logic (also uses &!?⊗⊕, which are elsewhere)
  | '□' | '◊' # modal logic
arithSymb ::=
    '±' | '∓' | '·' | '÷' | '√'
  | '‰' | '‱'
calculusSymb ::=
    '∂' | '∫' | '∇' | 'Δ'

mathOps ::=
    '∘' | '⨟' # composition
  | '⧺' # catenation
  | '⋆' | '∗' | '⋄'
  | '⊕' | '⊖' | '⊗' | '⊘' | '⊛' | '⊙' | '⊚'
  | '⊞' | '⊟' | '⊠' | '⊡' | '⊠' | '⊡' | '⧈'
  | '⋈' | '⋉' | '⋊'

mathRels ::= equivRels | compareRels
equivRels ::=
    '≠' | '≡' | '≢' # straight
  | '≃' | '≠' | '≅' | '≇' # semi-wiggly
  | '≁' | '≈' | '≉' # wiggly
  | '�given' | '≕' | '≔' # predicates
  | '≝' | '≙' | '≚' | '≔' | '≕' # define/assign
  | '≍' | '≭' | '≎' | '≏' # bendy
compareRels ::=
  # FIXME do I really need both handednesses of ≶≷⋚⋛?
    '≤' | '≥' | '≮' | '≯' | '≰' | '≱' # straight
  | '≺' | '≻' | '⩽' | '⩾' | '⊀' | '⊁' | '⪯̸' | '⪰̸'  # cusp
  | '⊏' | '⊐' | '⊑' | '⊒' | '⋤' | '⋥' # square
  | '≶' | '≷' | '≸' | '≹' | '≦' | '≧' # posets
  | '≲' | '≳' | '≴' | '≵' # approximate
  | '≪' | '≫' | '⋘' | '⋙' # multiple
```

```
mathTurnstiles ::=
    '⊢' | '⊨' | '⊩' | '⊫'

nArySymb ::=
    '⋃' | '⋂' | '⨿' | '⨯' # sets
  | '⋁' | '⋀' # logic
  | '∑' | '∏' | '⨋' # arithmetic
  | '⊕' | '⊗' | '⊙' # circled

arrows ::=
    '←' | '↑' | '→' | '↓' | '↔' | '↕' # straight
  | '↚'       | '↛'       | '↮'       # stroked
  | '⇐' | '⇑' | '⇒' | '⇓' | '⇔' | '⇕' # double
  | '⇍'       | '⇏'       | '⇎'       # stroked double
  # alternate bodies
  | '↜'       | '↝'       | '↭'       # wave
  | '⇜'       | '⇝'                   # squiggle
  # alternate heads
  | '↞' | '↟' | '↠' | '↡' # two headed
  | '↼' | '↾' | '⇀' | '⇂' # harpoons
  | '↤' | '↥' | '⇥' | '↧' # to bar
  | '∝' | '⍼' | '⊸' | '⍽' # sometimes used in linear logic as well
  # alternate tails
  | '↢'       | '↣'       # tail
  | '↩'       | '↪'       # hook
  | '↤' | '↥' | '↦' | '↧' # from bar
  | '⇤'       | '⇥'       # double from bar
  # only tails
  | '≺' | '≻' # single tail
  | '⋘' | '⋙' # double tail


unlikelyMath ::=
  # archaic and numeral greek
    '𐆊'…'𐆋' | 'ϗ'…'ϡ' | '𐅀'…'𐅴'
  # modified greek letters
  | '𝛛' | '𝜕'
  # extra dots
  | '⁘' | '⁙' | '⁖'
  # operators
  | '⨲' | '⨱' | '⨰' # plus and minus variants
  | '≀' # wreath product
  | '⦶' | '⦷' | '⦸' | '⦹'
  | '⦻' | '⦼' | '⦽'
  # equivalence relations
  | '⧡' | '⧢' | '⧣'
```

```
  # proper relations
  | '' | '' | '' | '' | '' | '' | '' | '' | ''
  # closed triangle relations
  | '' | '' | '' | '' | '' | '' | '' | ''
  # other relations
  | '' | '' | ''
  | '' # listed as "between", but I don't see how this is a trinary relation
  # math shapes (logic symbols have square and lozenge)
  | '' | '' | '○'
  | '' | ''
  | '' | '' | '' | ''
  | '' | '' # I've seen the last used in "SICP for GR" or whatever it's called
  # there are even more math symbols in Unicode!
  | moreUnlikelyMath

unlikelyArrows ::=
    '' | '' | '' | '' # triple body
  | '' | '' | '' | '' # quadruple body
  | '' | '' | '' | '' # arc (only down and left are available)
  | '' | '' | '' | '' | '' | '' | '' | '' # double harpoon
  | '' | '' | '' | '' # other-handed harpoons
  | ''
  | '' | '' # loop tail
  | '' | '' | '' | '' # fish tail
  | '' | '' # two headed from bar
  # turning
  | '' | '' | '' | '' | '' | '' | '' | '' # tip turn
  | '' | '' | '' | '' # curving
  | '' | '' | '' | '' # semicircle
  | '' | '' | '' | '' # (triangle-headed) u-shaped
  | '' | '' # circle
  | '' | '' # gapped circle
  | '' | '' # closed circle
  # multiple
  | '' | '' | '' | '' | '' | '' | '' | '' # paired
  | '' | '' | '' | '' # \ paired harpoons
  | '' | '' | '' | '' # |
  | '' | '' | '' | '' # |
  | '' | '' | '' | '' # /
  | '' | '' # three

moreUnlikelyMath ::=
# set theory
  ▯▯▯▯ # n-ary
  ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
  ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
```

```
# logic
  □□
  ⌐□□
  □□□□□□□□□□□□□□□□□□□□□□□□
  □□ # n-ary
# arithmetic
  □□□ # dotted minuses
# calculus
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□
# relations
  □□□□□□□□ # straight
  □□□□□□□ # wiggles
  □□□□□□□□□ # straight and wiggly
  □□□□□□□□□□ # dotted
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□
  □□□□
  □□□□□□□□□□□□□□□□□□
# operators
  □□
  □□□□□□□□□□□□□□□□
  □□□□
  □□□□□□□□
  □□
  □□
  □□ # I honestly have no idea what these are
# tacks
  □□
  □□
  □□
  □□
  □□□□
  □□
  □□□□□□□□
# database
  □□□□
  □□
# misc
  □ # QED/tombstone
  □
  □□□□□□□
```

Warning

If you use non-ASCII notation in your language or library, it is not enough to include suggestions for ASCII alternatives. Instead, there must be a

tool to convert losslessly between Unicode and ASCII.

Warning

The selection of available symbol characters is not yet set in stone. So far, I have only examined a portion of Unicode for useful characters. My goal was to identify mathematical operators/relations/symbols, arrows, greek letters, and fancy alphanumerics at least. If something you need is missing, it may simply be because I couldn't easily find why certain symbols are used; let me know and I'll try to add it. I may include a number of other scripts if it comes to it, but I am loathe to include characters from scripts I am not fluent with; if they are to be added, it should be in consultation with people who *do* understand these scripts.

Warning

I have not gone through the available symbols to handle visually-confusable characters. I expect I'll allow visually-identical symbols, but use some normalization algorithm to ensure they are treated identically and normalized.

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)

cStrUnit ::= cStrChar
           |  BS cEscapeChar
```

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

```
          |  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.

Rationale

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.

Rationale

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.

Rationale

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
```

Rationale

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.

Heredocs are multi-line strings that have no escaping whatsoever because their content is included verbatim. They begin with a line ending in a "triple quote" and optional identifier and end with a line beginning with that same identifier (if any) and another triple quote. All the intervening lines are included unaltered, including alternate newlines, control characters, and so on. Additionally, heredocs may also be indented by including a backslash after the opening identifier and another backslash to indicate the indent amount.

Indicating indent depth with a backslash on the first line of a heredoc allows for heredocs to start with leading whitespace. Accommodations have been made for aligning indented backslash despite the first-line backslash: when indentation is done with spaces, the backslash counts towards the number of spaces of leading indent, and when it is done with tabs, a tab must follow the backslash[5]. The final newline before the close quotes is not included in the heredoc, so if you would like a heredoc to end with a trailing newline, leave a blank line.

```
heredoc ::= startHeredoc(name, ws, n)
            heredocLine(name, ws, n)*
            endHeredoc(name, ws, n)

openHeredoc(\name) ::= DQ DQ DQ (?name = symbolChar*) lws(_, _)?
startHeredoc(name, _, 0) ::= openHeredoc(name) nl
```

[5]So don't set your tab size less than two. That's what we call "expert guidance"!

```
startHeredoc(name, SP, n + 1) ::= openHeredoc(name) BS lws(_, _)? nl
                                  zlws(SP, n) BS
startHeredoc(name, HT, n + 1) ::= openHeredoc(name) BS lws(_, _)? nl
                                  zlws(HT, n) BS HT

heredocLine(name, ws, n) ::= zlws(ws, n) heredocContent nl
heredocContent ::= (1 - nl)* - (closeHeredoc(name) (1 - nl)*)

endHeredoc(name, ws, 0) ::= closeHeredoc(name)
endHeredoc(name, ws, n) ::= lws(ws, n) closeHeredoc(name)
closeHeredoc(name) ::= name DQ DQ DQ

zlws(_, 0) ::= ε
zlws(ws, n) ::= lws(ws, n)
```

This makes them an attractive target for embedding non-eexpr content.

Rationale

Heredocs are an attractive target for embedding non-eexpr content. I especially want to use if for embedding documentation as doc strings rather than comments[6]. Writing documentation in comments makes it more difficult to attach them to language constructs. The difference can be seen very well in Python, which uses docstrings, and can therefore report documentation directly in the REPL, or during script execution. True, documentation could be written in an e-expr-based markup, but I'd likely prefer markdown—I'm not a zealot. Regardless, heredocs can embed any desired documentation language.

Another use for heredocs is embedding non-eexpr languages more generally. An obvious use would be for crafting SQL expressions directly in a web server. Other possibilities are embedding a scripting language such as Lua, or C code as part of a foreign function interface.

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) ::= '}'
```

[6] In fact, I've only recently gotten over pronouncing "heredoc" as "docstring".

```
  # TODO more enclosers
  # ⬚⬚⬚⬚⬚⬚[]⬚⬚⬚⬚⬚⬚⬚«»
  # ⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚
  # ⬚⬚
 # how to rewrite in ASCII? perhaps use fat colon + parens `⬚f x y⬚ → ap::(f x y) ≡ f <$> x <*> y`

semicolon ::= ';'
comma ::= ','

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

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 without tooling, the easier it is everywhere"[7]. 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 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.[8]

[7] 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.

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

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