SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 0(0), 1–30 (MONTH 1998)

Unparsing Expressions With Prefix and Postfix Operators

Norman Ramsey
Department of Computer Science, University of Virginia, Charlottesville, VA 22903

August 13, 1998

SUMMARY

Unparsing is the problem of transforming an internal representation of a program into an external, concrete syntax. In conjunction with prettyprinting, it is useful for generating readable programs from internal representations. If the target language uses prefix and postfix operators, the problem is nontrivial. This paper shows how to unparse expressions using a simple, bottom-up tree walk, which keeps track of the least tightly binding operator not enclosed by parentheses. During the tree walk, this operator is compared with the operator of the parent expression, and parentheses are inserted based on the precedence, associativity, and fixity (infix, prefix, or postfix) of the two operators. The paper is a literate program. It includes code for the unparser and for its inverse parser, both of which can handle operators of dynamically chosen precedence and associativity. Supporting such operators is useful for languages like ML, in which programmers may assign precedence and associativity to their own functions.

KEY WORDS

KEY WORDS Unparsing Parsing Prettyprinting Literate Programming Standard ML

1. INTRODUCTION

Automatic generation of computer programs from high-level specifications is a well-known technique, which has been used to create scanners, parsers, code generators, tree rewriters, packet filters, dialogue handlers, machine-code recognizers, and other tools. To make it easier to debug an application generator, and to help convince prospective users that the generated code can be trusted, the generated code should be as idiomatic and readable as possible. It is therefore desirable, and sometimes essential, that the generated code use infix, prefix, and postfix operators, and that expressions be presented without unnecessary parentheses. Another helpful technique is to use a prettyprinter to automate indentation and line breaking.[1,2] This paper presents a method of automatically parenthesizing expressions that use prefix, postfix, and implicit operators; the method is compatible with automatic prettyprinting. The paper also shows how to parse expressions that use such operators. The parsing algorithm can be used even if operator precedences are not known at compile time, so it can be used with arbitrarily many user-defined precedences.

This paper is a "literate program,"[3] which means that a single source file is used to prepare both manuscript and code. Including code provides a formal, precise, testable statement of the unparsing

CCC 0038–0644/98/000001–30
Received 29 August 1997
Revised 27 March 1998

algorithm. The code is written in the purely functional subset of Standard ML.[4] ML is a good choice for several reasons. Referential transparency makes the subset easy to reason about, so we can show the correctness of simple unparsing algorithms. It is possible to express some properties of the data as type properties, which are checked by the compiler. Finally, SML's parameterized modules make the code reusable in several different contexts—the code that appears in the paper is the code used in both the SLED and the λ-RTL compilers,[5,6] for example. The paper explains many ML constructs as they are used, but readers not familiar with ML may still wish to consult References 7–9 for an introduction.

The paper presents algorithms for two versions of the unparsing problem. In the first version, all operators are binary infix operators; this version of the unparsing problem is well understood, and it is simple enough to make a proof of correctness tractable. The invariants and insights from this proof lead to an algorithm for a more general version, which handles prefix and postfix operators, as well as non-associative infix operators of arbitrary arity. The algorithm can also handle some "mixfix" operators, like the array-indexing operator $\cdot[\cdot]$. The algorithm is the main contribution of this paper. It is presented as executable code, which uses parameterized modules to make it independent of the representation of the unparsed form.

2. PARSING AND UNPARSING

Parsers and unparsers translate between concrete and abstract syntax. The concrete syntax used to represent expressions is usually ambiguous; ambiguities are resolved using operator precedence and parentheses. Internal representations are unambiguous but not suitable as concrete syntax. For example, expressions are often represented as simple trees, in which an expression is either an "atom" (constant or variable) or an operator applied to a list of expressions. We can describe such trees by the following grammar, written in extended BNF,[10] in which nonterminals appear in *italic* and terminals in *slant* fonts:

$$\begin{array}{rcl} \textit{expression} & \Rightarrow & \textsl{atom} \\ & | & \textsl{operator}\ \{\textit{expression}\} \end{array}$$

LISP and related languages use a nearly identical concrete syntax, simply by requiring parentheses around each application. The designers of other languages have preferred other concrete syntax, using infix binary operators, prefix and postfix unary operators, and possibly "mixfix" operators, as described by the following grammar:

$$\begin{array}{rcl} \textit{expression} & \Rightarrow & \textsl{atom} \\ & | & \textit{expression}\ \textsl{infix-operator}\ \textit{expression} \\ & | & \textsl{prefix-operator}\ \textit{expression} \\ & | & \textit{expression}\ \textsl{postfix-operator} \\ & | & \textit{expression}\ \{\textsl{nary-operator}\ \textit{expression}\} \\ & | & \textit{other form of expression}\ldots \end{array}$$

Even if the internal form distinguishes these kinds of operators, this grammar is unsuitable for parsing, because it is ambiguous. Grammars intended for parsing use various levels of precedence and associativity to resolve ambiguities. Once precedence and associativity have been determined, they can be encoded by introducing new nonterminals for each level of precedence. Users of shift-reduce parsers can specify precedence and associativity directly, then have the parser use them to resolve shift-reduce conflicts.

The parsing problem is to translate from concrete syntax to internal representation. The unparsing problem is to take an internal representation and to produce a concrete syntax that, when parsed, reproduces the original internal representation. That is, an unparser is correct only if it is a right inverse of its parser. For example, if our internal representation is a tree representing the product of $x+y$ with z, we cannot simply walk the tree and emit the concrete representation $x+y\times z$; we must emit $(x+y)\times z$ instead.

As stated, the unparsing problem does not have a unique solution; in general, there are many possible concrete syntaxes that, when parsed, reproduce the original internal representation. For example, we could choose a concrete syntax that parenthesizes every expression, as in $(((x)+(y))\times(z))$. The LISP rule parenthesizes every application, as in $((x+y)\times z)$. Rules like these use too many parentheses. Code with extra parentheses is unidiomatic and unreadable, like this C code, which has been emitted using something like the LISP rule:

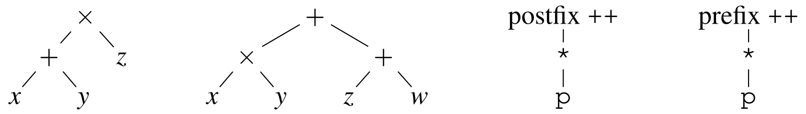
```
if (((Mem.u).Index8).index) != 4)
  if (!((unsigned(((Mem.u).Index8).d)) < 0x100))
    fail((("((Mem.u).Index8).d = 0x%x won't fit in 8 unsigned bits"),
           (((Mem.u).Index8).d)));
  else
    {
      emit((((7 | (15 << 4)) | (1 << 3)), 1));
      emit((((4 | (3 << 3)) | (1 << 6)), 1));
      emit((((((((Mem.u).Index8).ss) << 6) |
             ((((Mem.u).Index8).index) << 3)) |
             (((Mem.u).Index8).base)), 1));
      emit((((((Mem.u).Index8).d) & 0xff), 1));

    }
else
  fail(("Conditions not satisfied for constructor CALL.Epod"));
```

The unparser in this paper is not merely a parser's right inverse; it also produces results without extra parentheses:

```
if (Mem.u.Index8.index != 4)
  if (!((unsigned)Mem.u.Index8.d < 0x100))
    fail("Mem.u.Index8.d = 0x%x won't fit in 8 unsigned bits",
         Mem.u.Index8.d);
  else
    {
      emit(7 | 15 << 4 | 1 << 3, 1);
      emit(4 | 3 << 3 | 1 << 6, 1);
      emit(Mem.u.Index8.ss << 6 | Mem.u.Index8.index << 3 |
           Mem.u.Index8.base, 1);
      emit(Mem.u.Index8.d & 0xff, 1);
    }
else
  fail("Conditions not satisfied for constructor CALL.Epod");
```

The unparsing method uses information about precedence, associativity, and "fixity" of operators to transform an internal form into a concrete syntax. The method works from the bottom up and from the inside out. In each expression, it finds the operator, and it considers the subexpressions and their positions, left or right, relative to the operator. It decides whether to parenthesize subexpressions by comparing precedence and associativity of the current operator with the precedences and associativities of the most loosely binding operators in the subexpressions. Operators that are "covered" by parentheses do not participate in the comparison. Here are four example trees:



(a) (b) (c) (d)

In tree (a), discussed above, the unparser renders the left subtree as $x+y$, without parentheses, because the subexpressions x and y contain no operators. At the root, because operator $\times$ has higher precedence than operator $+$, the unparser parenthesizes the left subexpression, and the final result is $(x+y)\times z$. In tree (b), $+$ has lower precedence than $\times$, so it is not necessary to parenthesize the left subtree. Because $+$ is left-associative, however, it is necessary to parenthesize the right subtree, because it is a right child of a parent with the same precedence and left associativity. The final result is therefore $x\times y+(z+w)$. Trees (c) and (d) show prefix and postfix operators as used in C. In tree (c), `++` and `*` have opposite "fixity" (one is postfix and one is prefix), and because `++` has higher precedence than `*`, it is necessary to parenthesize the subexpression, resulting in `(*p)++`. In tree (d), `++` and `*` have the same fixity, so precedence is irrelevant, and the expression is unparsed as `++*p`.

Although the full unparsing method handles prefix, postfix, and infix operators, it is easier to understand if we begin with the simpler case in which all operators are binary infix operators.

3. PARSING AND UNPARSING WITH INFIX OPERATORS

Abstract and concrete syntax for infix operators

Type `rator` represents a binary infix operator, which has a text representation, a precedence, and an associativity:

⟨*infix*⟩≡

```
type precedence = int
datatype associativity = LEFT | RIGHT | NONASSOC
type rator = string * precedence * associativity
```

This ML code uses simple integers (`int`) to represent precedence, an enumeration to represent associativity, and a triple to represent an operator. (In the context of an ML type definition, a `*` does not represent multiplication; it connects elements of a tuple.) The more general unparser, presented below, shows how to use an arbitrary type, not just `string`, as an operator's concrete representation.

Precedence and associativity determine how infix expressions are parsed into trees, or equivalently, how they are parenthesized. For example, if operator $\otimes$ has higher precedence than operator $\oplus$, then $x\oplus y\otimes z = x\oplus(y\otimes z)$ and $x\otimes y\oplus z = (x\otimes y)\oplus z$. When two operators have the same precedence, associativity is used to disambiguate. If $\oplus$ is left-associative, $x\oplus y\oplus z = (x\oplus y)\oplus z$; if it is right-associative, $x\oplus y\oplus z = x\oplus(y\oplus z)$. Some languages have *non-associative* operators; if $\oplus$ is non-associative, then

$$(x\oplus y)\oplus z \neq x\oplus y\oplus z \neq x\oplus(y\oplus z),$$

and $x\oplus y\oplus z$ may be illegal. The comma that separates parameters in a C function call is an example of a non-associative operator.

Many expressions are obtained by applying operators to other expressions, but there must always be indivisible constituents of expressions. We call such constituents *atoms*. They appear at the leaves of abstract syntax trees and as the non-operator tokens in concrete syntax. In most languages the atoms include identifiers, integer and real literals, string literals, and so on. In our initial, simple model, an expression is either an atom or an infix operator applied to two expressions:

⟨*infix*⟩+≡
```
datatype ast = ATOM of string
             | APP  of ast * rator * ast
```

Type `ast` represents an expression's abstract syntax tree. This ML `datatype` definition is something like a production in a context-free grammar; it gives two alternative ways of constructing an `ast`. These alternatives are always given names (`ATOM` and `APP`), which are called *constructors*. An ML datatype may contain any number of constructors, and there may be data associated with each one. Here, for an atom, we care only about the atom's string representation. For an application, we want the operator and the `asts` representing the two operands. Because `ast` is used in its own definition, it is a recursive type, and values of type `ast` are trees.

Type type `ast` represents the input to the unparser; we also need a type to represent the unparser's output. For simplicity, we treat this output as a sequence of lexemes, where a lexeme represents an atom, an operator, or a parenthesis. Moreover, we undertake to emit only sequences in which atoms and operators alternate, or in which parenthesized sequences take the place of atoms. Let us call such a sequence an `image` and use a representation that forces it to satisfy the following grammar:

image ⇒ *lexical-atom* {operator *lexical-atom*}

lexical-atom ⇒ atom | (*image*)

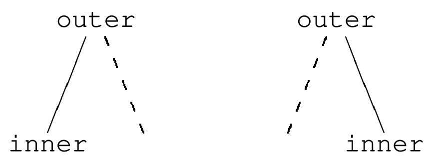
In the corresponding ML, the sequence in braces becomes the type `image'`:

⟨*infix*⟩+≡
```
datatype  image  = IMAGE of lexical_atom * image'
and       image' = EOI (* end of image *)
                 | INFIX of rator * lexical_atom * image'
and lexical_atom = LEX_ATOM of string
                 | PARENS of image
```

`EOI` represents the end of the image, and `PARENS` represents an image in parentheses. This representation enforces the invariants that expressions and operators alternate, and that the first and last elements of an image are expressions.

Parsing infix expressions

To be correct, an unparser must produce a sequence that parses back into the original abstract syntax tree. We develop an unparsing algorithm by thinking about parsing. To understand how to minimize parentheses, we need to consider where parentheses are needed to get the correct parse. Suppose we have an abstract syntax tree that has been obtained by parsing an image without any parentheses. Then wherever the syntax tree has an `APP` node whose parent is also an `APP` node, there are two cases: the child may be the left child or the right child of its parent:



Because this tree was obtained by parsing parenthesis-free syntax, either the inner operator has higher precedence than the outer, or they have the same precedence and associativity and the associativity is to the left (right) if the inner is a left (right) child of the outer. We formalize this condition using the predicate `noparens(inner, outer, `*side*`)`, where *side* is `LEFT` or `RIGHT` and `noparens` is defined as follows:

⟨*infix*⟩+≡ $\boxed{\texttt{noparens : rator * rator * associativity -> bool}}$

```
fun noparens (inner as (_, pi, ai), outer as (_, po, ao), side) =
          pi > po
    orelse pi = po andalso ai = ao andalso ao = side
```

where `pi`, `ai`, `po`, and `ao` stand for precedence or associativity of inner or outer operators.

This ML code defines a function `noparens`. Its name and type are given in the box at the beginning of the chunk. Type assertions in boxes are inserted into the code and checked by the compiler. The type of `noparens` shows that it is a Boolean function of three arguments (two operators and an associativity). The structure of the definition is `fun` *name binding* = *body*. The *binding* associates names to the arguments; these names are used in the body. `inner as (_, pi, ai)` is a nested binding; it binds `pi` to the precedence of `inner` and `ai` to the associativity of `inner`. The underscore `_` is a "wildcard," which indicates that the text representation of `inner` is to be ignored and not bound to any name. The process of using the *binding* to associate names with the arguments and parts of the arguments is called *ML pattern matching*.

Readers familiar with the treatment of operator-precedence parsing in Section 4.6 of Reference 11 may recognize that

$$\begin{aligned}\texttt{noparens}(i, o, \texttt{LEFT}) &\equiv i \cdot\!> o \\ \texttt{noparens}(i, o, \texttt{RIGHT}) &\equiv o <\!\cdot\, i\end{aligned}$$

We use `noparens` during unparsing to decide when parentheses are needed and during parsing to decide whether to shift or reduce.

To prove correctness of the unparser, I introduce a simple LR parser. The unparser is correct if `parse (unparse e) = e` for any syntax tree `e`. In other words, if we begin with an abstract syntax tree `e`, unparse it, and then parse the resulting lexemes, we get back the original abstract syntax tree.

In the parser's stack, expressions and operators alternate. Unless the stack is empty, there is an operator on the top, which is written to the right.

⟨*infix*⟩+≡

```
datatype stack = BOT
               | PUSH of stack * ast * rator
```

This recursive datatype is isomorphic to a list of (operator, ast) pairs, but it is helpful to define a special type because the constructors `BOT` and `PUSH` distinguish parser stacks from other kinds of lists.

The parser itself can be simplified by a bit of trickery. Instead of treating "bottom of stack" or "end of input" as special cases, treat them as occurrences of a sentinel operator, `minrator`. `minrator` has precedence `minprec`, which must be lower than the precedence of any real operator. Using this trick, we define functions that return the operator on the top of the stack and the operator about to be scanned in the input.

⟨*infix*⟩+≡

```
srator : stack  -> rator
irator : image' -> rator
```

```
val minrator : rator =
 ("<minimum-precedence sentinel>", minprec, NONASSOC)
fun srator (PUSH  (_, _, $)) = $
  | srator BOT               = minrator
fun irator (INFIX ($, _, _)) = $
  | irator EOI               = minrator
```

Given a stack `stack` and an input sequence of tokens `ipts`, we may abbreviate `srator stack` as $\oplus_{\texttt{stack}}$ and abbreviate `irator ipts` as $\oplus_{\texttt{ipts}}$.

These function definitions show a new ML construct—we define functions on `datatypes` by pattern-matching against the datatype constructors. We provide a *binding* and a *body* for each constructor. This ability to use pattern matching to do case analysis is one of the advantages of using ML, especially because the compiler ensures that all cases are covered.

This code also illustrates an unusual syntactic feature of ML: we can use strings of symbols, as well as strings of letters, to make identifiers. In this paper, by convention, the identifier `$` stands for an arbitrary operator.

We now have enough machinery to write a shift-reduce LR parser, which continues parsing until its stack and input are empty. The parser's state includes a stack with an operator $\oplus_{\texttt{stack}}$ on top, a current expression `e`, and the input, which begins with the operator $\oplus_{\texttt{ipts}}$. This state is represented by three parameters to the function `parse'`. Parsing is complete when the stack is empty (`BOT`) and the input is empty (`EOI`).

⟨*infix*⟩+≡

```
parse'    : stack * ast * image' -> ast
parse     : image -> ast
parseAtom : lexical_atom -> ast
```

```
exception Associativity
local
  exception Impossible
  fun parse' (BOT, e, EOI) = e
    | parse' (stack, e, ipts) =
        ⟨shift or reduce, depending on relative precedence of ⊕stack and ⊕ipts⟩
  and parse (IMAGE(a, tail)) = parse'(BOT, parseAtom a, tail)
  and parseAtom (LEX_ATOM a) = ATOM a
    | parseAtom (PARENS im) = parse im
in
  val parse = parse
end
```

The ML code uses two new constructs, and it shows a new way to use pattern matching. The declaration `local` *private-declarations* `in` *public-declarations* `end` hides information. Private declarations are invisible outside the `local` construct; only public declarations escape. The connective `and` is used with `fun` to define a nest of mutually recursive functions. The first match in the definition of `parse'` handles the case when both the stack and the input are empty. The next match handles all other cases; this works because a variable in a *binding* matches any datatype constructor.

The interesting part of the parser is deciding whether to shift or reduce. Shifting pushes `e` and $\oplus_{\texttt{ipts}}$ onto the stack. Reducing pops $\oplus_{\texttt{stack}}$ and the expression below it, creating a new `APP` node with operator $\oplus_{\texttt{stack}}$. Shifting $\oplus_{\texttt{ipts}}$ guarantees that $\oplus_{\texttt{ipts}}$ will be reduced before $\oplus_{\texttt{stack}}$, and therefore that $\oplus_{\texttt{ipts}}$ will eventually be a right descendant of $\oplus_{\texttt{stack}}$. Reducing guarantees that $\oplus_{\texttt{stack}}$ will eventually be a left descendant of $\oplus_{\texttt{ipts}}$. The parser uses `noparens` to choose whichever alternative is correct without parentheses. The choice is deterministic because the conditions for shifting and reducing are mutually exclusive; the proof of correctness relies on that fact.

⟨*shift or reduce, depending on relative precedence of* $\oplus_{\texttt{stack}}$ *and* $\oplus_{\texttt{ipts}}$⟩≡

```
if noparens(irator ipts, srator stack, RIGHT) then (* shift *)
  case ipts
    of INFIX ($, a, ipts') =>
         parse'(PUSH(stack, e, $), parseAtom a, ipts')
     | EOI => raise Impossible (* EOI has lowest precedence *)
else if noparens(srator stack, irator ipts, LEFT) then (* reduce *)
  case stack
    of PUSH (stack', e', $) => parse'(stack', APP(e', $, e), ipts)
     | BOT => raise Impossible (* BOT has lowest precedence *)
else
  raise Associativity
    (* saw a + b + c, for some nonassociative + *)
```

Pattern matching can be used in `case` statements as well as in function definitions. Here it is used to extract the first two inputs or the expression and operator on top of the stack. Because `srator` and `irator` return `minrator` for `BOT` and `EOI`, the parser shifts when the stack is empty and reduces when there are no more inputs. Therefore, the lines that raise `Impossible` are never executed; they are included only to keep the compiler from complaining about cases that are not covered.

The parser uses two different kinds of recursion. The direct recursive calls from `parse'` to itself are all tail calls; such calls are ML programmers' way of writing loops. In an imperative language, `parse'` would be written with a loop, and it would modify three variables: the stack, the current expression, and the inputs. When the stack and inputs were empty, it would exit the loop by returning the current expression. An imperative version of `parse'` would keep the indirect recursive call through `parseAtom` and `parse`.

Unparsing infix expressions

Before getting into the details of unparsing, we define functions that make images from atoms and put parentheses around images.

⟨*infix*⟩+≡

```
image       : string -> image
parenthesize : image  -> image
```

```
fun image        a  = IMAGE (LEX_ATOM a, EOI)
fun parenthesize im = IMAGE (PARENS im,  EOI)
```

`infixImage` combines two images by putting an infix operator between them. It uses `append` as an auxiliary function.

⟨*infix*⟩+≡

```
append     : image' * image' -> image'
infixImage : image * rator * image -> image
```

```
local
  fun append (EOI, image') = image'
    | append (INFIX($, a, tail), image') =
```

```
        INFIX($, a, append(tail, image'))
in
  fun infixImage (IMAGE(a, tail), $, IMAGE(a', tail')) =
        IMAGE(a, append(tail, INFIX($, a', tail')))
end
```

To unparse an expression, we produce a *fragment* containing not only the image of the expression, but also the lowest-precedence operator used in the expression. That operator tells us enough to decide whether to parenthesize that expression when it is used.

⟨*infix*⟩+≡

```
type fragment = rator * image
```

The *top-level operators* of an image are the operators that appear outside of parentheses. If an image has top-level operators, then the top-level operators of least precedence must all have the same associativity, which must be `LEFT` or `RIGHT`. Otherwise, one of those operators would have to be in parentheses, which contradicts the assumption that they are top-level operators.

The function `bracket(frag, side, rator)` parenthesizes a fragment `frag` that is to appear next to an operator `rator` on the side labelled `side`:

⟨*infix*⟩+≡ `bracket : fragment * associativity * rator -> image`

```
fun bracket((inner, image), side, outer) =
  if noparens(inner, outer, side) then image else parenthesize image
```

Given the ability to bracket fragments, unparsing is straightforward. We create another sentinel operator `maxrator`, having precedence `maxprec`, which must be higher than the precedence of any true operator, so we can use it in fragments made from atoms.

⟨*infix*⟩+≡

```
unparse' : ast -> rator * image
unparse  : ast -> image
```

```
local
  val maxrator = ("<max-precedence sentinel>", maxprec, NONASSOC)
  fun unparse' (ATOM a) = (maxrator, image a)
    | unparse' (APP(l, $, r)) =
        let val l' = bracket (unparse' l, LEFT,  $)
            val r' = bracket (unparse' r, RIGHT, $)
        in  ($, infixImage(l', $, r'))
        end
in
  fun unparse e =
    let val ($, im) = unparse' e
    in  im
    end
end
```

Using `bracket` maintains the precedence and associativity invariants of the fragments. `unparse` first computes a fragment, then discards the operator, leaving only the image.

Proof of correctness

Proving the simple unparser correct helps clarify how the parser and unparser work together. The result giving the most insight is Proposition 3, which gives the properties of the top-level operators produced during unparsing. Most of the proof is omitted from the body of the paper; it appears instead in Appendix B.

Proposition 1 `parse (unparse e)` $=$ `e`.

This proposition follows from Proposition 2.

Proposition 2 (Correctness of unparsing in context) *If* `e` *is any expression, and if* `a` *and* `tail` *are chosen such that* `unparse e = IMAGE(a, tail)`*, and if* `s` *is a parser stack and* `i` *an input sequence of type* `image'` *such that for all top-level operators* $\oplus'$ *in* `tail`*,*

$$\mathtt{noparens}(\oplus', \mathtt{srator\ s}, \mathtt{RIGHT}) \textit{ and } \mathtt{noparens}(\oplus', \mathtt{irator\ i}, \mathtt{LEFT}),$$

then `parse' (s, parseAtom a, append(tail, i)) = parse' (s, e, i)`.

In other words, even when `e` is a fragment of a larger expression, we can parse the concrete syntax `(a, tail)` after having parsed a prefix (stack `s`) and before parsing a suffix (inputs `i`), and in this context we recover the original expression `e`, leaving stack `s` and inputs `i` to be processed. The conditions involving `noparens` say that the top-level operators in `unparse e` bind at least as tightly as $\oplus_{\mathtt{stack}}$, which is on top of the stack, and at least as tightly as $\oplus_{\mathtt{ipts}}$, which follows `e`; furthermore, when the binding powers are the same, the associativity forces the parser to shift or reduce as appropriate.

Definition 1 (Binds as tightly as) *An operator* $\oplus$ binds as tightly as *another operator* `rator` *if and only if*

$$\begin{aligned} &\text{precedence}\,\oplus > \text{precedence}\,\mathtt{rator} \\ \vee\, &\text{precedence}\,\oplus = \text{precedence}\,\mathtt{rator} \wedge \text{associativity}\,\oplus = \text{associativity}\,\mathtt{rator}. \end{aligned}$$

This relation is like `noparens`, except it ignores the relative positions of operators of the same precedence and associativity. This weakening of `noparens` yields a reflexive, transitive, binary relation, which is in fact a slight variation on "has precedence at least as great."

The proof of Proposition 2 is by structural induction on `e`. The base case is straightforward. To get the idea of the proof of the induction step, we use textual juxtaposition to write expressions, inputs, and parser stacks. If $\mathtt{e} = \mathtt{l} \oplus \mathtt{r}$, we can write $\mathtt{unparse\ e} = \mathtt{a_l t_l} \oplus \mathtt{a_r t_r}$, where `a`'s are atoms and `t`'s (tails) are sequences of atoms and operators. Writing $\mathtt{parse'}(\mathtt{s}, \mathtt{parseAtom\ a_l}, \mathtt{t_l} \oplus \mathtt{a_r t_r i})$ for the state of the parser, we prove Proposition 2 by showing

$$\mathtt{parse'}(\mathtt{s}, \mathtt{parseAtom\ a_l}, \mathtt{t_l} \oplus \mathtt{a_r t_r i})$$

$=$ (by the induction hypothesis applied to `l`)

$$\mathtt{parse'}(\mathtt{s}, \mathtt{l}, \oplus \mathtt{a_r t_r i})$$

$=$ (by hypothesis, $\mathtt{noparens}(\oplus, \mathtt{srator\ s}, \mathtt{RIGHT})$, and the parser shifts)

$$\mathtt{parse'}(\mathtt{sl}\oplus, \mathtt{parseAtom\ a_r}, \mathtt{t_r i})$$

$=$ (by the induction hypothesis applied to `r`)

$$\texttt{parse}'(\texttt{sl}\oplus,\texttt{r},\texttt{i})$$

$=$ (by hypothesis, $\texttt{noparens}(\oplus,\texttt{irator i},\texttt{LEFT})$, and the parser reduces)

$$\texttt{parse}'(\texttt{s},\texttt{l}\oplus\texttt{r},\texttt{i}) = \texttt{parse}'(\texttt{s},\texttt{e},\texttt{i})$$

The details involve showing that the preconditions hold and it is safe to apply the induction hypothesis. We need facts about the results returned by `unparse` and `unparse'`, as stated in the next two propositions.

Proposition 3 *If an expression* $\texttt{e} = \texttt{APP}(\texttt{l},\oplus,\texttt{r})$*, then we can choose* $\texttt{l}'$ *and* $\texttt{r}'$ *as in* $\texttt{unparse}'$*, so that* $\texttt{unparse e} = \texttt{l}'\oplus\texttt{r}'$.

1. *If* $\oplus$ *is non-associative, then every top-level operator in* $\texttt{l}'$ *and* $\texttt{r}'$ *has a precedence greater than that of* $\oplus$.
2. *If* $\oplus$ *is left-associative, then every top-level operator in* $\texttt{r}'$ *has a precedence greater than that of* $\oplus$.
3. *If* $\oplus$ *is right-associative, then every top-level operator in* $\texttt{l}'$ *has a precedence greater than that of* $\oplus$.

Proposition 4 *For any expression* `e`*,* `unparse' e` *is properly covered, as defined below.*

Definition 2 (Covered, properly covered) *A fragment* `frag = (rator, im)` *is* covered *if and only if every top-level operator in* `im` *binds as tightly as* `rator`*. A covered fragment is* properly covered *if and only if* `rator = maxrator` *or there is at least one top-level operator in* `im`*.*

The intuition behind a covered fragment `(rator, im)` is that low-precedence operators in `im`, which do not bind as tightly as `rator`, are safely "covered" with parentheses. Proper coverage avoids pathology for fragments that don't have any top-level operators.

Proposition 3 is proved by case analysis on the result returned by `bracket`. Proposition 4 is proved by structural induction on the expression unparsed. The proof uses this lemma:

Lemma 1 (Bracketing lemma) *If fragment* `f` *is properly covered, then for any operator* `rator` *and associativity* `a`*,* `bf = (rator, bracket(f, a, rator))` *is covered.*

The lemma is proved by case analysis on whether `bracket` parenthesizes its fragment. The proof uses transitivity of "binds as tightly as."

The proof is not as simple as it would have been if `parse` and `unparse` had been designed in tandem with a proof, but `parse` and `unparse` were designed to have simple implementations, not a simple proof. One reason the proof is complex is that the functions are not full inverses; because parsing leaves no record of redundant parentheses, $\texttt{unparse}(\texttt{parse e}) \neq \texttt{e}$. Moreover, although the functions are partial inverses, the computations are not inverses. The parser is an LR parser, working left to right with an explicit stack. The unparser works not left to right but bottom-up, with an implicit stack created by recursive calls.

An alternative approach to unparsing might begin with a parser and attempt to derive an unparser using the program-transformation techniques known as *program inversion*.[12-14] If successful, this attempt would produce an unparser that would be correct by construction, but it would look substantially different from the one presented here. Program-inversion rules have been developed for simple programming calculi, not real programming languages, so the unparser would have to be translated into a programming language in order to be executed. The unparsers in this paper are executable directly, and they are designed to be easy to understand, even if that design makes the proof of correctness harder to understand.

4. ADDING PREFIX AND POSTFIX OPERATORS

Simple infix expressions are inadequate for many real programming languages, including C and ML. In this section, we add unary prefix and postfix operators, as well as infix operators of arbitrary arity. Moreover, we use the Standard ML modules system to generalize our idea of what an image is, so we can produce output for a prettyprinter, not just text.

Fixity distinguishes prefix and postfix operators from infix operators. Only infix operators have associativity.

⟨*assoc.sml*⟩≡
```
structure Assoc = struct
  datatype associativity = LEFT | RIGHT | NONASSOC
  datatype fixity = PREFIX | POSTFIX | INFIX of associativity
end
```
From these definitions, the possible values of type `fixity` are `PREFIX`, `POSTFIX`, `INFIX LEFT`, `INFIX RIGHT`, and `INFIX NONASSOC`.

Having prefix and postfix operators means that precedence alone no longer determines the correct parse, because the order of two prefix or of two postfix operators overrides precedence. For example, if `++` and `*` are prefix operators and `p` is an atom, then no matter what the precedences of `++` and `*`, `++*p` and `*++p` are both correct and unambiguous, equivalent to `++(*p)` and `*(++p)`, respectively.

Infix operators of arbitrary arity broaden our view of non-associative operators. For example, in both ML and C, the following three expressions are all legal, but all different:

$$\texttt{f}((\texttt{a, b}),\ \texttt{c}) \neq \texttt{f}(\texttt{a, b, c}) \neq \texttt{f}(\texttt{a},\ (\texttt{b, c}))$$

The comma operator is an n-ary, infix, non-associative operator.

Operators now have a concrete representation, a precedence, and a fixity.

⟨*general types*⟩≡
```
type precedence = int
type rator = Atom.atom * precedence * Assoc.fixity
```
In expressions, operators may be unary, binary, or n-ary.

⟨*type of expressions*⟩≡
```
datatype ast = ATOM   of Atom.atom
             | UNARY  of rator * ast
             | BINARY of ast * rator * ast
             | NARY   of rator * ast list
```
List types are built in to ML. Type `'a list` means "a list of values of type `'a`," for any type `'a`. It is convenient to treat `NARY(`$\oplus$`, [e])` as equivalent to `e` for any $\oplus$; `[e]` is the singleton list containing `e`.

In this new implementation, atomic components of expressions are no longer limited to strings, but may be any values of any type. That type is given the name `Atom.atom`, but it is not otherwise specified. The user of the unparser can pick any convenient type, e.g., prettyprinter directives. The structure `Atom` is a parameter to the unparser module. It must provide the following types and values:

⟨*properties of structure* `Atom`⟩≡
```
type atom (* representation of operator or atomic expression *)
val parenthesize : atom list -> atom
```
The unparser uses `Atom.parenthesize` to parenthesize a list of atoms.

The chunk ⟨*properties of structure* `Atom`⟩ exploits a feature of the Standard ML modules system; in a *signature*, we can name types and values without giving their definitions. The unparser module is parameterized by a module called `Atom`, and it works with *any* `Atom` that provides a type `atom`, a function `parenthesize`, and the other values listed in ⟨*properties of structure* `Atom`⟩. This parameterization makes the unparser general and easy to reuse.

To make this general capability as useful as possible, we make it possible to have the unparser add "decorations" to list of atoms. We do so by adding another kind of expression.

⟨*type of expressions*⟩+≡
```
              | DECORATE of (Atom.atom list -> Atom.atom) * ast
```
The `DECORATE` constructor attaches a "decoration function" to an expression. The decoration function must transform a list of atoms into a single atom in a way that is indetectible by the parser. For example, a decoration function might add values of type `Atom.atom` that indicate changes in indentation, color, or font of the unparsed text. These values would appear to the parser as white space.

Finally, the user of the unparser must supply minimum and maximum precedences, which are used to define the sentinels `minrator` and `maxrator`.

⟨*properties of structure* `Atom`⟩+≡
```
  val minprec : int (* precedence below that of any operator *)
  val maxprec : int (* precedence above that of any operator *)
  val bogus : atom (* bogus atom used to make sentinel operators *)
```
In many applications, `minprec` and `maxprec` can be computed automatically.

The datatype `ast` and the structure `Atom` form the external interfaces to the unparser. In the implementation, images are more general—operators and atomic terms no longer alternate, but they are still properly parenthesized:

$$\textit{image} \Rightarrow \{\text{operator} \mid \textit{lexical-atom} \mid \textit{decorated-image}\}$$

$$\textit{lexical-atom} \Rightarrow \text{atom} \mid \texttt{(}\ \textit{image}\ \texttt{)}$$

The corresponding ML types are:

⟨*general types*⟩+≡
```
  datatype  image  = IMAGE     of lexeme list
  and       lexeme = RATOR     of rator
                   | EXP       of lexical_atom
                   | DECORATED of (Atom.atom list -> Atom.atom) * image
  and lexical_atom = LEX_ATOM  of Atom.atom
                   | PARENS    of image
```
where `lexeme` corresponds to $(\text{operator} \mid \textit{lexical-atom} \mid \textit{decorated-image})$.

Type `image` is internal; the unparser returns a list of atoms. It uses `flatten` to convert an `image` into such a list. It uses `Atom.parenthesize` for parenthesization.

⟨*general*⟩≡

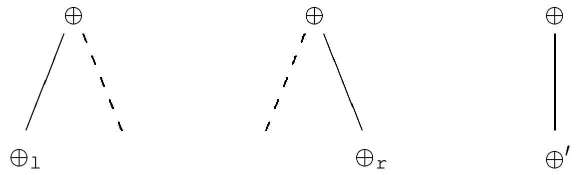
```
image        : image  -> Atom.atom list
lexeme       : lexeme -> Atom.atom
lexical_atom : lexical_atom -> Atom.atom
flatten      : image  -> Atom.atom list
```

```
local
  fun image (IMAGE l) = map lexeme l
  and lexeme (RATOR (atom, _, _)) = atom
    | lexeme (EXP a) = lexical_atom a
    | lexeme (DECORATED (f, im)) = f (image im)
  and lexical_atom (LEX_ATOM atom) = atom
    | lexical_atom (PARENS im) = Atom.parenthesize (image im)
in
  val flatten = image
end
```

Each local function computes an atom or list of atoms that corresponds to a value of one type. By programming convention, the name of each function is the name of the type it operates on. Because `local` hides these functions, `flatten` is the only name visible outside this code chunk.

Parenthesizing general expressions

As in the simpler case, development of an unparser begins with the invariants that apply to an abstract syntax tree produced by parsing an image without parentheses. There are now three cases to consider: an operator may be the left child of a binary infix operator, the right child of a binary infix operator, or the only child of a unary prefix or postfix operator.



(a) (b) (c)

As before, the rules for parenthesization are encapsulated in a `noparens` function. Because these trees are obtained by parsing parenthesis-free syntax, in case (a), noparens($\oplus_{\mathtt{l}}$,$\oplus$,LEFT); in case (b), noparens($\oplus_{\mathtt{r}}$,$\oplus$,RIGHT); and in case (c), noparens($\oplus'$,$\oplus$,NONASSOC).

⟨*general*⟩+≡

```
noparens : rator * rator * Assoc.associativity -> bool
```

```
fun noparens(inner as (_, pi, fi), outer as (_, po, fo), side) =
  pi > po orelse                              (* (a), (b), or (c) *)
  case (fi, side)
    of (POSTFIX,     LEFT)  => true           (* (a) *)
     | (PREFIX,      RIGHT) => true           (* (b) *)
     | (INFIX LEFT,  LEFT)  =>
         pi = po andalso fo = INFIX LEFT      (* (a) *)
     | (INFIX RIGHT, RIGHT) =>
         pi = po andalso fo = INFIX RIGHT     (* (b) *)
     | (_,           NONASSOC) => fi = fo     (* (c) *)
     | _ => false
```

These rules can be understood as follows:

- If the inner operator has higher precedence, parentheses are not needed.
- If a postfix operator appears to the left of an infix operator (a), it always applies to the preceding expression, and parentheses are not needed. Similarly when a prefix operator appears to the right of an infix operator (b).
- In case (a), if $\oplus_l$ has lower precedence than $\oplus$, then parentheses are needed, but if the two have the same precedence, then parentheses can be avoided, *provided* both are left-associative. A similar argument applies to case (b) when both operators are right-associative.
- Finally, in case (c), if the precedence of $\oplus'$ is not greater than the precedence of $\oplus$, then parentheses can be avoided if and only if both operators have the same fixity, i.e., both are prefix operators or both are postfix operators. This is the case in the example of `++*p` used to introduce this section.

These rules for parenthesization, as embodied in the `noparens` predicate, are the key to understanding both parsing and unparsing algorithms. The `noparens` function is used in an unparser, which appears below, and in a parser, which appears in Appendix A.

As before, the unparser needs auxiliary functions to manipulate images. Because an image is now simply a list of lexemes, these functions are a bit simpler.

⟨*general*⟩+≡

```
image        : Atom.atom -> image
parenthesize : image -> image
infixImage   : image * rator * image -> image
prefixImage  :         rator * image -> image
postfixImage : image * rator         -> image
```

```
fun image a            = IMAGE [EXP (LEX_ATOM a)]
fun parenthesize image = IMAGE [EXP (PARENS image)]
fun infixImage  (IMAGE l, $, IMAGE r) = IMAGE (l @ RATOR $ :: r)
fun prefixImage (         $, IMAGE r) = IMAGE (    RATOR $ :: r)
fun postfixImage(IMAGE l, $         ) = IMAGE (l @ RATOR $ :: [])
```

where `@`, `::`, and `[]` are list append, cons, and the empty list, which are all predefined in ML.

We use the same `bracket` to parenthesize sub-expressions.

⟨*general*⟩+≡

```
bracket : fragment * Assoc.associativity * rator -> image
```

```
type fragment = rator * image
fun bracket((inner, image), side, outer) =
  if noparens(inner, outer, side) then image else parenthesize image
```

The structure of the unparser is as before:

⟨*general*⟩+≡

```
unparse : ast -> Atom.atom list
```

```
local
  val maxrator = (Atom.bogus, Atom.maxprec, INFIX NONASSOC)
  exception Impossible
  ⟨function unparse', to map an expression to a fragment⟩
in
  fun unparse e =
    let val ($, im) = unparse' e
    in  flatten im
    end
end
```

The unparsing function itself has new cases: the unary operator, the *n*-ary operator, and the decorated expression. The unary operator appears before or after its operand, depending on its fixity.

⟨*function* `unparse'`, *to map an expression to a fragment*⟩≡ `unparse' : ast -> fragment`

```
fun unparse' (ATOM a) = (maxrator, image a)
  | unparse' (BINARY(l, $, r)) =
      let val l' = bracket (unparse' l, LEFT,  $)
          val r' = bracket (unparse' r, RIGHT, $)
      in  ($, infixImage(l', $, r'))
      end
  | unparse' (UNARY($, e)) =
      let val e' = bracket (unparse' e, NONASSOC, $)
          val (_, _, fixity) = $
      in  ($, case fixity of PREFIX  => prefixImage ($, e')
                           | POSTFIX => postfixImage(e', $)
                           | INFIX _ => raise Impossible)
      end
  | unparse' (NARY(_, []))    = raise Impossible
  | unparse' (NARY($, [e]))   = unparse' e
  | unparse' (NARY($, e::es)) =
      let val leftmost = bracket(unparse' e, LEFT, $)
          fun addOne(r, leftPrefix) =
                infixImage(leftPrefix, $,
                           bracket(unparse' r, RIGHT, $))
      in  ($, foldl addOne leftmost es)
      end
  | unparse' (DECORATE (f, e)) =
      let val ($, im) = unparse' e
      in  ($, IMAGE [DECORATED (f, im)])
      end
```

The interesting case for the *n*-ary operator is covered by the pattern `NARY($, e::es)`, which represents the operator `$` applied to more than one operand. The code inserts `$` between the operands. The first operand, `e`, is used to make `leftmost`, an initial "left prefix." The `addOne` function extends the left prefix, adding operator and operand on the right. The built-in `foldl` function walks the list `es`, using `addOne` to extend the left prefix with the remaining operands. Because both `LEFT` and `RIGHT` are used in the arguments to `bracket`, operator must be non-associative for the operands to be parenthesized properly. Of course, no other associativity makes sense, because a left- or right-associative operator can always be treated as an infix binary operator. The *n*-ary case is needed only for non-associative operators.

5. USING THE UNPARSER

Before the unparser can be used, it has to be instantiated with a suitable `Atom` module. Once the user creates an `Atom` module, he or she combines it with the unparser's implementation to create `Unparser`, an instance of the unparser that is specialized to that particular `Atom`. This instance provides the datatype `ast`, with its contructors `ATOM`, `UNARY`, etc. It also provides an unparsing function.

⟨*values exported by the unparser*⟩≡

```
val unparse : ast -> Atom.atom list
```

Once the unparser is instantiated, it can be used in a real tool, where unparsing is usually one of a series of transformations. In a typical application generator, a program being generated passes through several representations:

- an application-specific internal representation,
- an unparsing tree of type `Unparser.ast`,
- a list of atoms of type `Atom.atom list`, which typically includes prettyprinting directives as well as text, and
- the final program, represented as nicely indented text in the target language.

This section discusses transformation from an internal representation into an unparsing tree, using examples from the New Jersey Machine-Code Toolkit.[15] The first part of this paper discusses transformation from an unparsing tree to a list of atoms. References 1 and 2 discuss transformation of prettyprinting directives to indented text.

A preliminary step in creating unparsing trees is assigning precedence and associativity to the operators of the target language. This can be done easily and reliably by putting operators in a data structure that indicates precedence implicitly and fixity (which includes associativity) explicitly. A fragment of the structure used for C is

```
    ⋮
(L, [">>", "<<"]),
(L, ["+", "-"]),
(L, ["%", "/", "*"]),
    ⋮
```

where `L` is an abbreviation for `Assoc.INFIX Assoc.LEFT`, “infix left” being one of the five possible fixities. Precedence increases as we move down the structure. This data structure is transformed into functions `Cprec` and `Cfixity` that return the precedence and fixity of given operators. It is also used to compute the values of `minprec` and `maxprec`, which are needed to create an unparser.

Supplying `minprec` and `maxprec` to the unparser may seem awkward. One could supply instead a list of all operators to be used, letting the unparser compute `minprec` and `maxprec`. This technique seems unnecessarily restrictive; it would limit the unparser not only to a single kind of atom, but also to a single set of operators. As written, one unparser can be (and is) used to emit code in different target languages, using different operator sets.

Another way to eliminate `minprec` and `maxprec` would be to use a different representation of precedence inside the unparser, e.g., to represent precedence by an ML datatype that included values $+\infty$ and $-\infty$ in addition to all the integers. In a production environment, eliminating `minprec` and `maxprec` from the interface would justify a more complex implementation. For expository purposes, however, another user-defined datatype and special comparison functions would distract readers from the explanation of unparsing.

The fixities of operators can be surprising. For example, the C operators `.` and `->`, which are used to select members from structures and unions, are superficially binary infix operators, but their right-hand “arguments” are not expressions but identifiers, and semantically they are unary operators. The Toolkit in fact treats them as unary postfix operators, choosing their representations dynamically according to what member is being selected. For example, to create an unparsing tree corresponding to the selection of member `tag` from structure `e`, it uses the function

⟨*tree-creation functions*⟩≡ `dot : Unparser.ast * string -> Unparser.ast`

```
fun dot (e, tag) =
  Unparser.UNARY((pptext ("." ^ tag), Cprec ".", Assoc.POSTFIX), e)
```

Here the ML ^ operator is string concatenation. The `pptext` function takes a string and converts it to an `Atom.atom`, making it intelligible to the prettyprinter.

Handling mixfix operators may involve unparsing some trees during the construction of other trees. For example, to translate an array-indexing expression into C, one must put the subscript in square brackets. The subscript is unparsed independently and turned into an atom by applying `pplist`. It is then wrapped in square brackets, and the indexing operation itself is represented by the empty string, but with proper precedence and associativity, which, for example, enable the unparser to produce either `*p[n]` or `(*p)[n]` as required. A similar trick is used to unparse function calls after applying an *n*-ary comma operator to the arguments.

⟨*tree-creation functions*⟩+≡

```
subscript : Unparser.ast * Unparser.ast -> Unparser.ast
```

```
fun subscript(array, index) =
      let val index = pplist (Unparser.unparse index)
          val index = pplist [pptext "[", index, pptext "]"]
      in  Unparser.BINARY ( array
                          , (pptext "", Cprec "subscript", L)
                          , Unparser.ATOM index
                          )
      end
```

`pplist` concatenates atoms; it is also used to define `Atom.parenthesize`:

```
fun parenthesize l = pplist [pptext "(", pplist l, pptext ")"]
```

6. USE AND AVAILABILITY

The unparser described in this paper is used in two application generators.[6, 15] Because the paper is a literate program, the code used in the generators is extracted directly from the source text of the paper, and it is available from the author's Web site, `http://www.cs.virginia.edu/~nr`. Practitioners who can't use this code directly can build unparsers by implementing the `noparens` predicate and writing `unparse` as a bottom-up tree walk. Appendix A of this paper presents the inverse of the general unparser, i.e., a parser that handles prefix and postfix as well as infix operators, all of dynamically chosen precedence and associativity. This parser, which is used in one of the application generators, is also available online. Appendix B presents the full proof of correctness of the simple unparser.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grant ASC-9612756 and by the US Department of Defense under contract MDA904-97-C-0247 and DARPA order number E381. The views expressed are my own and should not be taken to be those of the DoD.

I thank Referee A for demanding improvements in presentation.

APPENDIX: A. A PARSER FOR GENERAL EXPRESSIONS

The parser shown here is the inverse of the unparser in the body of the paper. It handles prefix, postfix, and infix operators. It also supports concrete syntax in which concatenation of two expressions signifies the application of a "juxtaposition" operator to the two expressions. Like other binary operators, juxtaposition has precedence and associativity. For example, in Standard ML, juxtaposition denotes function application, it associates to the left, and it has a precedence higher than that of any explicit infix operator.[4] In awk, juxtaposition denotes string concatenation, and it has a precedence higher than the comparison operators but lower than the arithmetic operators.[16]

The parser handles juxtaposition by inserting an implicit "juxtaposition operator" whenever two expressions are juxtaposed in the input. (The unparser handles juxtaposition by using a binary infix operator whose representation is white space.) The user supplies a specification of this operator as part of the structure `Atom`. The specification includes an associativity, not a fixity, because juxtaposition is infix by definition.

⟨*properties of structure* `Atom`⟩+≡
```
val juxtapositionSpec : atom * int * Assoc.associativity
```
Internally, the parser converts this specification into an infix operator.

⟨*general parser*⟩≡
```
local
  val (rep, prec, assoc) = Atom.juxtapositionSpec
in
  val juxtarator = (rep, prec, Assoc.INFIX assoc)
end
```
The parser's main data structure is a stack of operators and expressions satisfying these invariants:

1. No postfix operator appears on the stack.
2. If a binary operator is on the stack, its left argument is below it.
3. An expression on the stack is preceded by a list of prefix operators, then either an infix operator or the bottom of the stack.

These invariants are only partially enforced by the type system. All operators have the same type, so `prefixop` and `infixop` are only hints.

⟨*general parser*⟩+≡
```
type infixop   = rator
type prefixop  = rator
datatype stack = BOT
               | BIN of stack * prefixop list * ast * infixop
type stack_top = (prefixop list * ast)
```
The sentinel operator `minrator` appears on the bottom of the stack:

⟨*general parser*⟩+≡ $\boxed{\texttt{srator : stack -> rator}}$
```
val minrator = (Atom.bogus, Atom.minprec, INFIX NONASSOC)
fun srator  (BIN (_, _, _, $)) = $
  | srator  BOT = minrator
```

The parser's input looks something like this:

$$\textit{image} \Rightarrow \{\textit{prefix}\}\textit{lexical-atom}\{\textit{postfix}\}\ \{[\textit{infix}]\ \{\textit{prefix}\}\textit{lexical-atom}\{\textit{postfix}\}\}$$

In parsing, we assume that we can distinguish prefix, postfix, and infix operators. If juxtaposition is forbidden, these restrictions can be relaxed somewhat; it is then sufficient to be able to distinguish postfix from infix operators. The adjustments needed are left as an exercise. `isPrefix` identifies prefix operators.

⟨*general parser*⟩+≡ `isPrefix : rator -> bool`

```
fun isPrefix (_, _, f) = f = PREFIX
```

The parser uses two functions, `parsePrefix` and `parsePostfix`, depending on whether it is scanning before or after an atomic input. The overall structure of the parser is

⟨*general parser*⟩+≡ `parse : image -> ast`

```
exception ParseError of string * rator list
local
  exception Impossible
  ⟨general parsing functions⟩
in
  val parse = parse
end
```

There are many cases. `parsePrefix` maintains a list of prefix operators, which is saves until it reaches an atomic input. It is not safe to reduce the prefix operators, because the atomic input might be followed by a postfix operator of higher precedence. If `parsePrefix` encounters a non-prefix operator or end of file, the input is not correct. When `parsePrefix` encounters an atomic input, it passes the input and the saved prefixes to `parsePostfix`.

⟨*general parsing functions*⟩≡

`parsePrefix : stack * prefixop list * lexeme list -> ast`

```
fun parsePrefix(stack, prefixes, RATOR $ :: ipts') =
      if isPrefix $ then
        parsePrefix(stack, $ :: prefixes, ipts')
      else
        raise ParseError("%s is not a prefix operator", [$])
  | parsePrefix(_, _, []) =
      raise ParseError ("premature end of expression", [])
  | parsePrefix(stack, prefixes, EXP a :: ipts') =
      parsePostfix(stack, (parseAtom a, prefixes), ipts')
  | parsePrefix(stack, prefixes, DECORATED (_, IMAGE l) :: ipts') =
      parsePrefix(stack, prefixes, l @ ipts')
```

If it encounters decorated lexemes, `parsePrefix` ignores the decoration.

In addition to the stack and the input, `parsePostfix` maintains a current expression `e`, and a list `prefixes` containing unreduced prefix operators that immediately precede `e`. The idea is that operators arriving in the input must be tested either against the nearest unreduced prefix operator, or if there are no unreduced prefix operators, against the operator on the top of the stack. In the first case, the nearest unreduced prefix operator is called `$`, and it may be compared with an operator `irator` from the input. If `$` has higher precedence, it is reduced, by using `UNARY` to apply it to the current expression `e`, and it becomes the new expression. Otherwise, the parser's behavior depends on the fixity of `irator`; it may reduce `irator`, shift it, or insert `juxtarator`.

⟨*general parsing functions*⟩+≡

```
parsePostfix : stack * (ast * prefixop list) * lexeme list -> ast
```

```
and parsePostfix(stack, (e, $ :: prefixes),
                 ipts as RATOR (irator as (_, _, ifixity)) :: ipts') =
      if noparens($, irator, NONASSOC) then (* reduce prefix $ *)
        parsePostfix(stack, (UNARY($, e), prefixes), ipts)
      else if noparens(irator, $, NONASSOC) then
        (* irator has higher precedence *)
        case ifixity
          of POSTFIX => (* reduce postfix rator *)
               parsePostfix(stack,
                            (UNARY(irator, e), $ :: prefixes),
                            ipts')
           | INFIX _ => (* shift, look for prefixes *)
               parsePrefix(BIN(stack, $ :: prefixes, e, irator),
                           [], ipts')
           | PREFIX  => ⟨insert juxtarator⟩
      else
        raise ParseError
            ("can't parse (%s e %s); " ^
             "operators have equal precedence", [$, irator])
```

If `parsePostfix` encounters an atom in the input, it inserts `juxtarator` no matter what the state of unreduced prefixes, since consuming atoms is the purview of `parsePrefix`.

⟨*general parsing functions*⟩+≡

```
    | parsePostfix(stack, (e, prefixes), ipts as EXP _ :: _) =
        ⟨insert juxtarator⟩
```

The insertion itself is straightforward.

⟨*insert* `juxtarator`⟩≡

```
  parsePostfix(stack, (e, prefixes), RATOR juxtarator :: ipts)
```

The other major case for `parsePostfix` occurs when there are no more unreduced prefix operators. In that case, comparison must be made with `srator stack`, the operator on top of the stack.

⟨*general parsing functions*⟩+≡
```
    | parsePostfix(stack, (e, prefixes as []),
                   ipts as RATOR (irator as (_, _, ifixity)) :: ipts'
                 ) =
        if noparens(srator stack, irator, LEFT) then
          (* reduce infix on stack *)
          case stack
            of BIN (stack', prefixes, e', $) =>
                 parsePostfix(stack', (BINARY(e', $, e), prefixes),
                              ipts)
             | BOT => raise Impossible (*  BOT has lowest prec *)
        else if noparens(irator, srator stack, RIGHT) then
          case ifixity
            of POSTFIX => (* reduce postfix rator *)
                 parsePostfix(stack, (UNARY(irator, e), []), ipts')
             | INFIX _ => (* shift, look for prefixes *)
                 parsePrefix(BIN(stack, [], e, irator), [], ipts')
             | PREFIX  => ⟨insert juxtarator⟩
        else
           raise ParseError ("%s is non-associative", [irator])
```

As above, decorations are ignored.

⟨*general parsing functions*⟩+≡
```
    | parsePostfix (stack, (e, prefixes),
                    DECORATED (_, IMAGE l) :: ipts') =
        parsePostfix(stack, (e, prefixes), l @ ipts')
```

When the input is exhausted, the parser reduces first the prefixes, then the stack, until it reaches the result.

⟨*general parsing functions*⟩+≡
```
    | parsePostfix(stack, (e, $ :: prefixes), []) =
        (* reduce prefix $ *)
        parsePostfix(stack, (UNARY($, e), prefixes), [])
    | parsePostfix(BIN (stack', prefixes, e', $), (e, []), []) =
        (* reduce stack *)
        parsePostfix(stack', (BINARY(e', $, e), prefixes), [])
    | parsePostfix(BOT, (e, []), []) = e
```

We complete the parser with functions that parse an entire input, or an input in parentheses.

⟨*general parsing functions*⟩+≡

```
parse : image -> ast
parseAtom : lexical_atom -> ast
```

```
  and parse(IMAGE(l)) = parsePrefix(BOT, [], l)
  and parseAtom (LEX_ATOM a) = ATOM a
    | parseAtom (PARENS im)  = parse im
```

APPENDIX: B. PROOFS OF CORRECTNESS

Proofs are written in a goal-directed, calculational style. A goal-directed proof begins with a conclusion to be proved, then reasons backwards towards premises. The backwards arrow $\Leftarrow$ is pronounced "is implied by."

Proposition 1 `parse (unparse e)` = `e`.

Proof Rewrite `parse (unparse e)` into an expression more easily proved equal to `e`.

```
parse (unparse e)
```

= (by choosing suitable `a` and `tail` for the result of `unparse e`)

```
parse (IMAGE (a, tail))
```

= (by the definition of `parse`)

```
parse' (BOT, parseAtom a, tail)
```

= (by the definition of `append`)

```
parse' (BOT, parseAtom a, append(tail, EOI))
```

= (by Proposition 2)

```
parse'(BOT, e, EOI)
```

= (by definition of `parse'`)

```
e
```

$\diamond$

Proposition 2 (Correctness of unparsing in context) *If* `e` *is any expression, and if* `a` *and* `tail` *are chosen such that* `unparse e = IMAGE(a, tail)`, *and if* `s` *is a parser stack and* `i` *an input sequence of type* `image'` *such that for all top-level operators* $\oplus'$ *in* `tail`,

$$\texttt{noparens}(\oplus', \texttt{srator s}, \texttt{RIGHT}) \textit{ and } \texttt{noparens}(\oplus', \texttt{irator i}, \texttt{LEFT}),$$

then `parse' (s, parseAtom a, append(tail, i)) = parse' (s, e, i)`.

Proof By structural induction on `e`. If `e` is an atom, write `e = ATOM x`, and

```
parse' (s, parseAtom a, append(tail, i))
```

= (because by assumption `unparse e = IMAGE (LEX_ATOM x, EOI)`)

```
parse' (s, parseAtom (LEX_ATOM x), append(EOI, i))
```

= (by definitions of `parseAtom` and `append`)

```
parse' (s, ATOM x, i)
```

= (by assumption)

```
parse' (s, e, i)
```

For the induction step, $\mathtt{e}$ must have the form $\mathtt{e} = \mathtt{APP}(\mathtt{l}, \oplus, \mathtt{r})$, and we can choose $\mathtt{l}'$ and $\mathtt{r}'$ as in $\mathtt{unparse}'$, so $\mathtt{unparse}\ \mathtt{e} = \mathtt{infixImage}(\mathtt{l}', \oplus, \mathtt{r}')$. Because both $\mathtt{l}'$ and $\mathtt{r}'$ have type $\mathtt{image}$, we can write $\mathtt{l}' = \mathtt{IMAGE}(\mathtt{a_l}, \mathtt{t_l})$ and $\mathtt{r}' = \mathtt{IMAGE}(\mathtt{a_r}, \mathtt{t_r})$. By using juxtaposition to abbreviate appending, we write $\mathtt{unparse}\ \mathtt{e} = \mathtt{a_l t_l} \oplus \mathtt{a_r t_r}$.[1] We therefore consider

$$\mathtt{parse}'(\mathtt{s}, \mathtt{parseAtom}\ \mathtt{a_l}, \mathtt{t_l} \oplus \mathtt{a_r t_r i})$$

= (by applying the induction hypothesis to $\mathtt{l}'$, as justified below)

$$\mathtt{parse}'(\mathtt{s}, \mathtt{l}, \oplus \mathtt{a_r t_r i})$$

= (by definition of $\mathtt{parse'}$, the parser shifts, because by hypothesis, $\oplus$ is a top-level operator from $\mathtt{unparse}\ \ \mathtt{e}$ and therefore $\mathtt{noparens}(\oplus, \mathtt{srator}\ \mathtt{s}, \mathtt{RIGHT})$)

$$\mathtt{parse}'(\mathtt{sl}\oplus, \mathtt{parseAtom}\ \mathtt{a_r}, \mathtt{t_r i})$$

= (by applying the induction hypothesis to $\mathtt{r}'$, as justified below)

$$\mathtt{parse}'(\mathtt{sl}\oplus, \mathtt{r}, \mathtt{i})$$

= (by definition of $\mathtt{parse'}$, the parser reduces, because by hypothesis, $\oplus$ is a top-level operator from $\mathtt{unparse}\ \ \mathtt{e}$ and therefore $\mathtt{noparens}(\oplus, \mathtt{irator}\ \mathtt{i}, \mathtt{LEFT})$)

$$\mathtt{parse}'(\mathtt{s}, \mathtt{APP}(\mathtt{l}, \oplus, \mathtt{r}), \mathtt{i})$$

= (by assumption)

$$\mathtt{parse}'(\mathtt{s}, \mathtt{e}, \mathtt{i})$$

It remains to justify applying the induction hypothesis to $\mathtt{l}'$ and $\mathtt{r}'$. By definition of $\mathtt{unparse}$ and $\mathtt{bracket}$, either

$$\mathtt{l}' = \mathtt{unparse}\ \mathtt{l} \qquad \text{or} \qquad \mathtt{l}' = \mathtt{parenthesize}(\mathtt{unparse}\ \mathtt{l}).$$

In the latter case, $\mathtt{l}' = \mathtt{IMAGE}(\mathtt{PARENS}(\mathtt{unparse}\ \mathtt{l}), \mathtt{EOI})$, so by the definitions of $\mathtt{a_l}$ and $\mathtt{t_l}$, we have $\mathtt{a_l} = \mathtt{PARENS}(\mathtt{unparse}\ \mathtt{l})$ and $\mathtt{t_l} = \mathtt{EOI}$, and

$$\mathtt{parse}'(\mathtt{s}, \mathtt{parseAtom}\ \mathtt{a_l}, \mathtt{t_l} \oplus \mathtt{a_r t_r i})$$

= (by assumption and by definition of $\mathtt{parseAtom}$)

$$\mathtt{parse}'(\mathtt{s}, \mathtt{parse}(\mathtt{unparse}\ \mathtt{l}), \oplus \mathtt{a_r t_r i})$$

= (by the induction hypothesis applied to $\mathtt{l}$, with empty stack and inputs)

$$\mathtt{parse}'(\mathtt{s}, \mathtt{l}, \oplus \mathtt{a_r t_r i})$$

which is the step made above.

The case where $\mathtt{l}' = \mathtt{unparse}\ \mathtt{l}$ is the interesting case.

$$\mathtt{parse}'(\mathtt{s}, \mathtt{parseAtom}\ \mathtt{a_l}, \mathtt{t_l} \oplus \mathtt{a_r t_r i})$$

= (For all top-level operators $\oplus'$ in $\mathtt{t_l}$, $\mathtt{noparens}(\oplus', \oplus, \mathtt{LEFT})$, and we apply the induction hypothesis with the inputs $\oplus \mathtt{r}'\ \mathtt{i}$, concluding that)

* This shorthand abbreviates $\mathtt{unparse}\ \mathtt{e} = \mathtt{IMAGE}(\mathtt{a_l}, \mathtt{append}(\mathtt{t_l}, \mathtt{INFIX}(\oplus, \mathtt{a_r}, \mathtt{t_r})))$.

$\texttt{parse}'(\texttt{s},\texttt{parseAtom}\,\texttt{a}_\texttt{l},\texttt{t}_\texttt{l} \oplus \texttt{a}_\texttt{r}\texttt{t}_\texttt{r}\texttt{i})$

We must show $\texttt{noparens}(\oplus',\oplus,\texttt{LEFT})$ for any top-level $\oplus'$ from $\texttt{t}_\texttt{l}$. If $\oplus$ is non- or right-associative, then by Proposition 3, precedence$\oplus' >$ precedence$\oplus$, and $\texttt{noparens}(\oplus',\oplus,\texttt{LEFT})$ follows. If $\oplus$ is left-associative, then

$\texttt{noparens}(\oplus',\oplus,\texttt{LEFT})$

$\Leftarrow$ (by definition of noparens and left-associativity of $\oplus$)

$\oplus'$ binds as tightly as $\oplus$

$\Leftarrow$ (transitivity)

$\oplus'$ binds as tightly as $\oplus_l \wedge \oplus_l$ binds as tightly as $\oplus$

Where $\texttt{unparse}'\,\texttt{l} = (\oplus_\texttt{l},\texttt{a}_\texttt{l}\texttt{t}_\texttt{l})$ and the two clauses follow by applying Proposition 4 to l and e. $\diamond$

Proposition 3 *If an expression* $\texttt{e} = \texttt{APP}(\texttt{l},\oplus,\texttt{r})$*, then we can choose* $\texttt{l}'$ *and* $\texttt{r}'$ *as in* $\texttt{unparse}'$*, so that* $\texttt{unparse}\,\texttt{e} = \texttt{l}' \oplus \texttt{r}'$.

1. *If $\oplus$ is non-associative, then every top-level operator in $\texttt{l}'$ and $\texttt{r}'$ has a precedence greater than that of $\oplus$.*
2. *If $\oplus$ is left-associative, then every top-level operator in $\texttt{r}'$ has a precedence greater than that of $\oplus$.*
3. *If $\oplus$ is right-associative, then every top-level operator in $\texttt{l}'$ has a precedence greater than that of $\oplus$.*

Proof First, do case analysis on $\texttt{l}' = \texttt{bracket}(\texttt{unparse}'\,\texttt{l},\texttt{LEFT},\oplus)$. If $\texttt{l}'$ is parenthesized, then (3) and half of (1) hold vacuously because $\texttt{l}'$ has no top-level operators. Otherwise, we can assume that $\texttt{unparse}'\,\texttt{l} = (\oplus_\texttt{l},\texttt{l}')$, and

For all top-level operators $\oplus'$ in $\texttt{l}'$, precedence$\oplus' >$ precedence$\oplus$

$\Leftarrow$ (introducing $\oplus_\texttt{l}$)

For all top-level operators $\oplus'$ in $\texttt{l}'$, precedence$\oplus' \geq$ precedence$\oplus_l$
$\wedge$ precedence$\oplus_\texttt{l} >$ precedence$\oplus$

$\Leftarrow$ (definitions of “covered” and “binds as tightly as”)

Fragment $(\oplus_\texttt{l},\texttt{l}')$ is covered $\wedge$ precedence$\oplus_\texttt{l} >$ precedence$\oplus$

$\Leftarrow$ (definition of noparens)

Fragment $(\oplus_\texttt{l},\texttt{l}')$ is covered
$\wedge\ \texttt{noparens}((\oplus_\texttt{l},\texttt{l}'),\oplus,\texttt{LEFT}) \wedge$ associativity$\oplus \neq \texttt{LEFT}$

$\Leftarrow$ (definition of bracket)

Fragment $(\oplus_\texttt{l},\texttt{l}')$ is covered
$\wedge\ \texttt{bracket}((\oplus_\texttt{l},\texttt{l}'),\oplus,\texttt{LEFT})$ is not parenthesized $\wedge$ associativity$\oplus \neq \texttt{LEFT}$

$=$ (by assumption)

unparse l is covered
$\wedge\ \texttt{l}'$ is not parenthesized $\wedge$ associativity$\oplus \neq \texttt{LEFT}$

$\Leftarrow$ (Proposition 4, assumption)

associativity $\oplus \neq$ `LEFT`

Which proves (3) and half of (1). A similar argument proves (2) and the other half of (1). $\diamond$

Proposition 4 *For any expression* `e`, `unparse' e` *is properly covered.*

Proof By structural induction. The base case is trivial.
For the induction step, we must prove

`unparse' e` is properly covered

$=$ (letting $\mathtt{e} = \mathtt{l} \oplus \mathtt{r}$ and $\mathtt{unparse}\ \mathtt{e} = \mathtt{l}' \oplus \mathtt{r}'$ as above)

$(\oplus, \mathtt{l}' \oplus \mathtt{r}')$ is properly covered

$\Leftarrow$ ($\oplus$ is a top-level operator of $\mathtt{l}' \oplus \mathtt{r}'$)

$(\oplus, \mathtt{l}' \oplus \mathtt{r}')$ is covered

$=$ (definition of "covered")

For all top-level operators $\oplus'$ in $\mathtt{l}' \oplus \mathtt{r}', \oplus'$ binds as tightly as $\oplus$

$=$ (distributing)

For all top-level operators $\oplus'$ in $\mathtt{l}', \oplus'$ binds as tightly as $\oplus$
$\wedge$ $\oplus$ binds as tightly as $\oplus$
$\wedge$ For all top-level operators $\oplus'$ in $\mathtt{r}', \oplus'$ binds as tightly as $\oplus$

$=$ (reflexivity of "binds as tightly as," definition of "covered")

$(\oplus, \mathtt{l}')$ is covered $\wedge$ $(\oplus, \mathtt{r}')$ is covered

$=$ (body of `unparse'`)

$(\oplus, \mathtt{bracket}(\mathtt{unparse}'\ \mathtt{l}, \mathtt{LEFT}, \oplus))$ is covered
$\wedge$ $(\oplus, \mathtt{bracket}(\mathtt{unparse}'\ \mathtt{r}, \mathtt{RIGHT}, \oplus))$ is covered

$\Leftarrow$ (bracketing lemma)

$\mathtt{unparse}'\ \mathtt{l}$ properly covered and $\mathtt{unparse}'\ \mathtt{r}$ properly covered

which is the induction hypothesis. $\diamond$

Lemma 1 (Bracketing lemma) *If fragment* `f` *is properly covered, then for any operator* `rator` *and associativity* `a`, `bf = (rator, bracket(f, a, rator))` *is covered.*

Proof By case analysis on the result of `bracket`. If the result is parenthesized, then `bf` is trivially covered, because it contains no top-level operators.
Otherwise, let us write `f = ($, im)`. Our goal is to show that

`bf = (rator, bracket(($, im), a, rator)` is covered.

$\Leftarrow$ (by the assumption that the result of `bracket` is `im`) `(rator, im)` is covered

$\equiv$ (by Definition 2)

For all top-level operators $\oplus$ in `im`, $\oplus$ binds as tightly as `rator`

$\Leftarrow$ (by transitivity of "binds as tightly as")

For all top-level operators $\oplus$ in `im`, $\oplus$ binds as tightly as `$`
$\wedge$ `$` binds as tightly as `rator`

$\Leftarrow$ (pulling second clause out of $\forall$ and strengthening it)

(For all top-level operators $\oplus$ in `im`, $\oplus$ binds as tightly as `$`)
$\wedge$ `$` binds as tightly as `rator` $\wedge$ associativity `$` $=$ `a`

$\equiv$ (by definitions of "covered" and `noparens`)

`($, im)` is covered
$\wedge$ `noparens($, rator, a)`

$\Leftarrow$ (by definition of "properly covered" and of `bracket`)

`($, im)` is properly covered
$\wedge$ `bracket(($, im),` associativity `$, rator)` is not parenthesized

both of which are true by assumption. $\diamond$

REFERENCES

1. Derek C. Oppen, 'Prettyprinting', *ACM Transacations on Programming Languages and Systems*, **2**(4), 465–483 (1980).
2. John Hughes, 'The Design of a Pretty-printing Library', J. Jeuring and E. Meijer (eds.), *Advanced Functional Programming*. Springer Verlag, LNCS 925, 1995, pp. 53–96.
3. Donald E. Knuth, 'Literate programming', *The Computer Journal*, **27**(2), 97–111 (1984).
4. Robin Milner, Mads Tofte, and Robert W. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts, 1990.
5. Norman Ramsey and Mary F. Fernández, 'Specifying representations of machine instructions', *ACM Transactions on Programming Languages and Systems*, **19**(3), 492–524 (1997).
6. Norman Ramsey and Jack W. Davidson, 'Machine descriptions to build tools for embedded systems', *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*. Springer Verlag, June 1998, pp. 172–188.
7. Jeffrey D. Ullman, *Elements of ML Programming*, Prentice Hall, Englewood Cliffs, NJ, 1994.
8. Laurence C. Paulson, *ML for the working programmer*, Cambridge University Press, New York, second edition, 1996.
9. Matthias Felleisen and Daniel P. Friedman, *The Little MLer*, MIT Press, December 1997.
10. Niklaus Wirth, 'What can we do about the unnecessary diversity of notation for syntactic definitions?', *Communications of the ACM*, **20**(11), 822–823 (1977).
11. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
12. David Gries, *The Science of Programming*, Springer-Verlag, 1981.
13. Wei Chen and Jan Tijmen Udding, 'Program inversion: More than fun!', *Science of Computer Programming*, **15**(1), 1–13 (1990).

14. J. von Wright, ‘Program inversion in the refinement calculus’, *Information Processing Letters*, **37**(2), 95–100 (1991).
15. Norman Ramsey and Mary F. Fernández, ‘The New Jersey Machine-Code Toolkit’, *Proceedings of the 1995 USENIX Technical Conference*, New Orleans, LA, January 1995, pp. 289–302.
16. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, MA, 1988.