Unparsing Expressions With Prefix and Postfix Operators

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

August 25, 1997

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 they be presented without unnecessary parentheses. Another helpful technique is to use a prettyprinter to automate indentation and line breaking (Oppen 1980; Hughes 1995). 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, which means that it can be used with an arbitrary number of user-defined precedences.

This paper is a literate program (Knuth 1984); a single source file is used to prepare both the manuscript and some parsing and unparsing code. Including code provides a formal, precise, testable statement of the unparsing algorithm. The code is written in the purely functional subset of Standard ML. I've chosen to use ML 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, I use SML's parameterized modules to make the code reusable in several different contexts—the code that appears in the paper is the code used in the SLED compiler, for example. Readers not familiar with ML may wish to consult Ullman (1994) or Paulson (1991) for an introduction.

The paper presents two versions of the unparsing problem. The first part handles infix operators only; this case is simple enough that we can prove the correctness of the unparsing algorithm. We use the invariants and insights from this proof to create an algorithm for a more general version, which handles prefix and postfix operators, as well as non-associative infix operators of arbitrary arity. The code for this version uses parameterized modules, which makes 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 directly 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 (Wirth 1977), in which nonterminals appear in *italic* and terminals in *slant* fonts:

expression ⇒ *atom*
| *operator* {*expression*}

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:

expression ⇒ *atom*
| *expression infix-operator expression*
| *prefix-operator expression*
| *expression postfix-operator*
| *expression* {*nary-operator expression*}
| *other form of expression...*

Even if the internal form distinguishes these kinds of operators, this grammar is unsuitable for parsing, because it is wildly ambiguous. Grammars intended for parsing use various levels of precedence and associativity to disambiguate. 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 decide 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 three and emit the concrete representation $x+y\times z$; we must emit $(x+y)\times z$ instead. We can produce correct concrete syntax by following the LISP rule and placing parentheses around every nonterminal *expression*, but the results are unidiomatic and unreadable. To get readable results, we should use information about precedence, associativity, and "fixity" of operators to unparse an internal form into concrete syntax. This paper presents a general method of unparsing expressions with infix, prefix, and postfix operators. It also supports a concrete syntax in which concatenation of two expressions in the concrete syntax can be taken to signify the application of a "juxtaposition" operator to the two expressions, and juxtaposition can be given a 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 (Milner, Tofte, and Harper 1990). In awk, juxtaposition denotes string concatenation, and it has a precedence higher than the comparison operators but lower than the arithmetic operators (Aho, Kernighan, and Weinberger 1988). The method described in this paper can also be used to handle some mixfix operators by tinkering with the definition of operator.

3 Parsing and unparsing with infix operators

Abstract and concrete syntax for infix operators

We begin by tackling the simpler case in which all operators are binary infix operators. Type `rator` represents such an operator, which has a text representation, a precedence, and an associativity:

⟨*infix*⟩≡

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

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.

We use the type `atom` to refer to the atomic constituents of expressions. Atoms appear at the leaves of abstract syntax trees and as the non-operator

tokens in concrete syntax. In most languages such constituents include identifiers, integer and real literals, string literals, and so on. An expression is either an atom or an operator applied to two expressions:

⟨*infix*⟩+≡
```
datatype ast = AST_ATOM of atom
             | APP of ast * rator * ast
```

Type `ast` represents an expression's abstract syntax tree.

An unparser takes a syntax tree and produces source code. For simplicity, we treat source code 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. I call such a sequence an `image`, and I use a representation that forces it to satisfy the following grammar:

$$\textit{image} \Rightarrow \textit{lexical-atom}\{\textit{operator}\,\textit{lexical-atom}\}$$

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

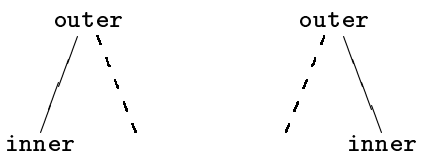
Here is the corresponding ML, in which the sequence in braces becomes the type `image'`:

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

`EOI` represents the end of the input, and `PARENS` represents an image in parentheses. This representation enforces the invariants that expressions and operators must 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 will 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 right 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 as follows:

⟨*infix*⟩+≡
```
fun noparens(inner as (_, pi, ai) : rator, outer as (_, po, ao), side) =
  pi > po orelse
  pi = po andalso ai = ao andalso ai = side
```

where `pi`, `ai`, `po`, and `ao` stand for precedence or associativity of inner or outer operators. Readers familiar with the treatment of operator-precedence parsing in Section 4.6 of Aho, Sethi, and Ullman (1986) 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 an operator-precedence parser. The correctness condition is that for any syntax tree `e`,

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

The parser uses an auxiliary stack in which expressions and operators alternative. Unless the stack is empty, there is an operator on top.

⟨*infix*⟩+≡
```
datatype stack = BOT
               | RATOR of rator * ast * stack
```

Before giving the parser itself, I introduce a simplifying trick. Instead of treating "bottom of stack" or "end of input" as special cases, I pretend they are occurrences of a phony operator `minrator`. `minrator` has precedence `minprec`, which must be lower than the precedence of any real operator. Using this trick, I can define functions that return the operator on the top of the stack and the operator about to be scanned in the input.

⟨*infix*⟩+≡
```
val minrator = ("<phony minimum-precedence operator>", minprec, NONASSOC)
fun srator  (RATOR ($, _, _)) = $
  | srator  BOT = minrator
fun irator  (INFIX ($, _, _)) = $
  | irator  EOI = minrator
```

In the ML code, I use the identifier `$` to stand for an arbitrary operator. Given a stack `stack` and an input sequence of tokens `ipts`, we will sometimes write $\oplus_s$ for `srator stack` and $\oplus_i$ for `irator ipts`.

We now have enough information to write an operator-precedence parser, which continues parsing until the stack and input are both empty. The state maintained by the parser includes a stack with an operator $\oplus_s$ on top, a current expression `e`, and the input, which begins with the operator $\oplus_i$. At each step, the parser may shift `e` and $\oplus_i$ onto the stack, or it may reduce the stack, consuming $\oplus_s$ and the expression below it. Reducing $\oplus_s$ creates a new `APP` node with operator $\oplus_s$, and it will eventually be a left descendant of $\oplus_i$. Shifting $\oplus_i$ guarantees that $\oplus_i$ will be reduced before $\oplus_s$, and that it will eventually be a right descendant of $\oplus_s$. We choose whichever alternative is correct without parentheses.

⟨*infix*⟩+≡

```
exception Associativity (* raised if a + b + c and + is nonassociative *)
local
  exception Impossible
  fun parse' (BOT, e, EOI) = e
    | parse' (stack, e, ipts) =
        if noparens(srator stack, irator ipts, LEFT) then (* reduce *)
          case stack
            of RATOR ($, e', stack') => parse'(stack', APP(e', $, e), ipts)
             | BOT => raise Impossible (* BOT has lowest precedence *)
        else if noparens(irator ipts, srator stack, RIGHT) then (* shift *)
          case ipts
            of INFIX ($, a, ipts') => parse'(RATOR($, e, stack), parse_atom a, ipts')
             | EOI => raise Impossible (* EOI has lowest precedence *)
        else
          raise Associativity
  and parse (IMAGE(a, tail)) = parse'(BOT, parse_atom a, tail)
  and parse_atom (LEX_ATOM a) = AST_ATOM a
    | parse_atom (PARENS im) = parse im
in
  val parse = parse
end
```

Making `srator` and `irator` return `minrator` for `BOT` and `EOI` guarantees that we always shift when the stack is empty and reduce when there are no more inputs. The conditions for shifting and reducing are mutually exclusive; we exploit that fact in the proof of correctness.

Unparsing infix expressions

Before getting into the details of unparsing, we had best show how to make images from atoms and how to put parentheses around an image to make it "lexically atomic."

⟨*infix*⟩+≡

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

We also have to be able to combine two images by putting an operator between them:

⟨*infix*⟩+≡
```
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* that contains not only the image of the expression, but also the lowest-precedence operator used in the expression. That operator tells us everything we need to know to decide whether to parenthesize that expression when it is used.

We define the *top-level operators* of an image to be those operators that appear outside of parentheses. If an image has any 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. We use `bracket(frag, side, rator)` to parenthesize a fragment `frag` that is to appear next to an operator `rator`, on the side labelled `side`:

⟨*infix*⟩+≡
```
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 phony 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*⟩+≡
```
local
  val maxrator = ("<maximum-precedence operator>", maxprec, NONASSOC)
  fun unparse' (AST_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
```

The use of `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 is not intrinsically interesting, but it helps to formalize our insight about how the parser and unparser work together. The most important part is Proposition 2, which gives the properties of the top-level operators produced during unparsing.

We begin with a lemma stating that the operator produced with `parse'` accurately reflects what's in the accompanying image.

Definition 1 *A fragment* `frag = (rator, im)` *is* covered *if and only if for all top-level operators* $\oplus$ *in* `im`*,*

$$\text{prec}\,\oplus \geq \text{prec}\,\texttt{rator} \wedge \text{prec}\,\oplus = \text{prec}\,\texttt{rator} \Rightarrow \text{assoc}\,\oplus = \text{assoc}\,\texttt{rator}.$$

A covered fragment is tight *if and only if there is at least one top-level operator in* `im` *or* `rator = maxrator`*.*

The intuition behind a covered fragment `(rator, im)` is that `im` can safely appear next to `rator` without parentheses.

We can now show that bracketing and unparsing preserve tightness.

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

Proof Let `f = ($, im)`. If the result of `bracket` is parenthesized, then `bf` is trivially covered. Otherwise, `noparens($, rator, a)` holds, which means that

$$\text{prec}\,\texttt{\$} \geq \text{prec}\,\texttt{rator} \wedge \text{prec}\,\texttt{\$} = \text{prec}\,\texttt{rator} \Rightarrow \text{assoc}\,\texttt{\$} = \text{assoc}\,\texttt{rator} = \texttt{a}.$$

Since `f` is tight, for any top-level operator $\oplus_{\texttt{im}}$ in `im`, $\text{prec}\,\oplus_{\texttt{im}} \geq \text{prec}\,\texttt{\$} \geq \text{prec}\,\texttt{rator}$. Moreover, if $\text{prec}\,\oplus_{\texttt{im}} = \text{prec}\,\texttt{rator}$, then both are equal to $\text{prec}\,\texttt{\$}$, and by the tightness of `f` they all have the same associativity. $\diamond$

Proposition 1 *For any expression* `e`*,* `unparse' e` *is tight.*

Proof By structural induction. The base case is trivial.

For the induction step, we let $\texttt{e} = \texttt{l} \oplus \texttt{r}$, and we prove tightness first.

The induction hypothesis and the bracketing lemma tell us that fragments $(\oplus, \texttt{l}')$ and $(\oplus, \texttt{r}')$ are covered. By reasoning like that used in the bracketing lemma, the fragment $(\oplus, \texttt{infixImage}(\texttt{l}', \oplus, \texttt{r}'))$ is covered. Since $\oplus$ is a top-level operator in $\texttt{infixImage}(\texttt{l}', \oplus, \texttt{r}')$, the fragment is tight. $\diamond$

Proposition 2 *If an expression* $\texttt{e} = \texttt{APP}(\texttt{l}, \oplus, \texttt{r})$*, then choose* $\texttt{l}'$ *and* $\texttt{r}'$ *so we can write* $\texttt{unparse e} = \texttt{l}' \oplus \texttt{r}'$*.*

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* $\oplus$*.*

Proof Follows from tightness and from the definition of `noparens`, which forces parenthesization unless top-level operators with the precedence of $\oplus$ are left-associative in $\texttt{l}'$ and right-associative in $\texttt{r}'$. $\diamond$

We prove `parse (unparse e) = e` by structural induction on the expression `e`. To make the induction work, we'll have to prove something a bit more elaborate.

Proposition 3 (Correctness of unparsing) *Let* `e` *be any expression. Choose* `a` *and* `tail` *such that we can write* `IMAGE(a, tail)` *for* `unparse e`*. Let* `s` *be a stack and* `i` *be an image sequence of type* `image'` *such that, for any top-level operator* $\oplus'$ *in* `tail`*,*

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

Then `parse' (s, parse_atom a, append(tail, i)) = parse'(s, e, i)`*.*

`parse (unparse e) = e` follows immediately by letting `s = BOT` and `i = EOI`.

Proof If `e` is an atom, we can write `e = AST_ATOM a`, and the proof is trivial:

```
unparse (ATOM a) = IMAGE (LEX_ATOM a, EOI)

parse' (s, exp (LEX_ATOM a), append(EOI, i)) =
parse' (s, ATOM a, i) = parse'(s, e, i)
```

For the induction step, `e` must have the form $\texttt{e} = \texttt{APP}(\texttt{l}, \oplus, \texttt{r})$, and we can choose $\texttt{l}'$ and $\texttt{r}'$ such that $\texttt{unparse e} = \texttt{l}' \oplus \texttt{r}'$. Note that either

$$\texttt{l}' = \texttt{unparse l} \quad \text{or} \quad \texttt{l}' = \texttt{parenthesize}(\texttt{unparse l}).$$

In the latter case, $\texttt{l}' = \texttt{IMAGE}(\texttt{PARENS}(\texttt{unparse l}), \texttt{EOI})$, and

$$\begin{aligned} \texttt{parse}'(\texttt{s}, \texttt{exp}(\texttt{PARENS}(\texttt{unparse l})), \texttt{i}) &= \texttt{parse}'(\texttt{s}, \texttt{parse}(\texttt{unparse l}), \texttt{i}) \\ &= \texttt{parse}'(\texttt{s}, \texttt{l}, \texttt{i}) \end{aligned}$$

by the induction hypothesis, for any `s` and `i`. A similar argument applies to $\texttt{r}'$. We can therefore apply the induction hypothesis to $\texttt{l}'$ and $\texttt{r}'$ without worrying if they have been parenthesized.

Let us write $\texttt{im} = \texttt{unparse e} = \texttt{l}' \oplus \texttt{r}'$ and $\texttt{l}' = \texttt{IMAGE}(\texttt{a}_\texttt{l}, \texttt{t}_\texttt{l})$. Now consider $\texttt{P} = \texttt{parse}'(\texttt{s}, \texttt{exp a}_\texttt{l}, \texttt{t}_\texttt{l} \oplus \texttt{r}'\ \texttt{i})$, and let $\oplus'$ be any top-level operator in $\texttt{t}_\texttt{l}$. If $\oplus$ is non- or right-associative, then by Proposition 2, $\mathrm{prec}\,\oplus' > \mathrm{prec}\,\oplus$, and therefore $\texttt{noparens}(\oplus', \oplus, \texttt{LEFT})$. If $\oplus$ is left-associative, then by tightness $\texttt{noparens}(\oplus', \oplus, \texttt{LEFT})$. In either case we can apply the induction hypothesis by letting new $\texttt{i}' = \oplus\texttt{r}'\ \texttt{i}$, concluding that $\texttt{P} = \texttt{parse}'(\texttt{s}, \texttt{l}, \oplus\texttt{r}'\ \texttt{i})$. By hypothesis, $\texttt{noparens}(\oplus, \texttt{srator s}, \texttt{RIGHT})$, so the parser shifts, and if $\texttt{r}' = \texttt{IMAGE}(\texttt{a}_\texttt{r}, \texttt{t}_\texttt{r})$ we have $\texttt{P} = \texttt{parse}'(\oplus\texttt{l s}, \texttt{exp a}_\texttt{r}, \texttt{t}_\texttt{r}\ \texttt{i})$. Again we argue by case analysis on the associativity of $\oplus$ that for any top-level operator $\oplus'$ in $\texttt{t}_\texttt{r}$, $\texttt{noparens}(\oplus', \oplus, \texttt{RIGHT})$. We reapply the induction hypothesis, this time letting new $\texttt{s}' = \oplus\texttt{l s}$, concluding that $\texttt{P} = \texttt{parse}'(\oplus\texttt{l s}, \texttt{r}, \texttt{i})$. By hypothesis, $\texttt{noparens}(\oplus, \texttt{irator i}, \texttt{LEFT})$, the parser reduces, and we have

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

$\diamond$

4 Adding prefix, postfix, and juxtaposition

Unparsing simple infix expressions is straightforward, but simple infix expressions are inadequate for many real programming languages, including C and ML. In this section, we add unary prefix and postfix operators, we add infix operators of arbitrary arity, and we permit the juxtaposition of two expressions to stand for the implicit binary operator `juxtarator`. 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 as well as just text.

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 $\sum$ and $\prod$ are prefix operators and $\vec{a}$ is an atom, then no matter what the precedences of $\sum$ and $\prod$, $\sum \prod \vec{a}$ and $\prod \sum \vec{a}$ are both correct and unambiguous, equivalent to $\sum(\prod \vec{a})$ and $\prod(\sum \vec{a})$, respectively.

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

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

We can handle this situation by treating the comma operator as an n-ary infix operator.

These changes lead to a new representation of expressions, in which operators may be unary, binary, or n-ary.

⟨*type of expressions*⟩≡

```
datatype ast = AST_ATOM of Atom.atom
             | UNARY    of rator * ast
             | BINARY   of ast * rator * ast
             | NARY     of rator * ast list
```

It will be convenient to treat `NARY(`$\oplus$`, [e])` as equivalent to `e` for any $\oplus$.

In this more general implementation, the atomic components of expressions are not limited to strings, but may be any value of type `Atom.atom`. The structure `Atom` need not be given at compile time; it is a parameter to the unparser module, and its only obligation is to provide the following types and values:

⟨*properties of structure* `Atom`⟩≡
```
type atom
val parenthesize : atom list -> atom
```

The unparser takes an abstract syntax tree and produces a concrete representation of type `atom list`; if it needs to parenthesize such a list, it uses the function `Atom.parenthesize`.

The user of the unparser can pick any convenient representation, e.g., strings or prettyprinter directives. If juxtaposition is permitted, the user must also supply `juxtarator`, the operator that is used implicitly when two expressions are juxtaposed.

⟨*properties of structure* `Atom`⟩+≡
```
type precedence = int
val juxtarator : atom * precedence * Assoc.associativity (* infix juxtaposition *)
```

Finally, we ask the user of the unparse to supply minimum and maximum precedences, so we can continue to use the phony `minrator` and `maxrator` as sentinels.

⟨*properties of structure* `Atom`⟩+≡
```
val minprec : precedence (* precedence below that of any operator *)
val maxprec : precedence (* precedence above that of any operator *)
val bogus : atom (* bogus atom used to construct phony operators *)
```

We use the new notion of *fixity* to distinguish 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
```

Operators have a concrete representation, a precedence, and a fixity. Juxtaposition is always infix.

⟨*general types*⟩≡
```
type precedence = int
type rator = Atom.atom * precedence * fixity
val juxtarator = let val (t, p, a) = Atom.juxtarator in (t, p, INFIX a) end : rator
```

In images, operators and atomic terms no longer alternate, but they are still properly parenthesized:

$$\textit{image} \Rightarrow \{\textit{operator} | \textit{lexical-atom}\}$$

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

The corresponding ML types are:

⟨*general*⟩≡
```
datatype  image  = IMAGE of lexeme list
and       lexeme = EXP   of lexical_atom
                 | RATOR of rator
and lexical_atom = LEX_ATOM of Atom.atom
                 | PARENS   of image
```

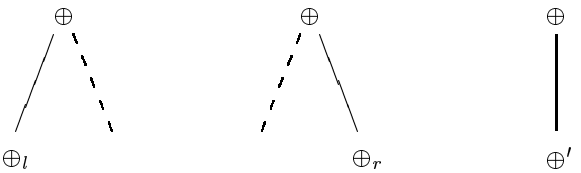
where `lexeme` corresponds to $(\textit{operator} | \textit{lexical-atom})$.

The unparser should return a list of atoms, so here is a function that converts an image into a list of atoms, using `Atom.parenthesize` for parenthesization.

⟨*general*⟩+≡
```
local
  fun image (IMAGE l) = map lexeme l
  and lexeme (EXP a) = lexical_atom a
    | lexeme (RATOR (atom, _, _)) = atom
  and lexical_atom (LEX_ATOM atom) = atom
    | lexical_atom (PARENS im) = Atom.parenthesize (image im)
in
  val flatten = image
end
```

Parenthesizing general expressions

As in the simpler case, our strategy for developing an unparser begins with the invariants that apply to an abstract syntax tree produced by parsing an image without any parentheses. Here there are three cases to consider: an operator is 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)

Because these trees are obtained by parsing parenthesis-free syntax, in case (a), `noparens`$(\oplus_1, \oplus, \texttt{LEFT})$; in case (b), `noparens`$(\oplus_\texttt{r}, \oplus, \texttt{RIGHT})$; and in case (c),

`noparens`($\oplus', \oplus$, `NONASSOC`). As before, the rules for parenthesization are encapsulated in a `noparens` function.

⟨*general*⟩+≡

```
fun noparens(inner as (_, pi, fi) : rator, 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 never needed.
- If a postfix operator appears to the left of an infix operator (a), it always applies to the preceding expression, and parentheses are never 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 no 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 $\sum \prod \vec{a}$ 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 is given here, 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 simplified.

⟨*general*⟩+≡

```
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)
```

where `@` is ML's built-in append operator.

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

⟨*general*⟩+≡

```
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*⟩+≡
```
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 and the n-ary operator. The unary operator appears before or after its operand, depending on its fixity.

⟨*function* `unparse'`*, to map an expression to a fragment*⟩≡
```
fun unparse' (AST_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 empty = IMAGE []
          val (_, _, fixity) = $
      in  ($, if fixity = PREFIX then infixImage(empty, $, e')
              else                    infixImage(e', $, empty))
      end
  | unparse' (NARY(_, []))    = raise Impossible
  | unparse' (NARY($, [e]))   = unparse' e
  | unparse' (NARY($, e::es)) =
      let val leftmost' = bracket(unparse' e, LEFT, $)
          fun addOne(r, l') = infixImage(l', $, bracket(unparse' r, RIGHT, $))
        in  ($, foldl addOne leftmost' es)
        end
```

The case for the n-ary operator inserts the operator `$` between the 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, this is the only case that makes sense, since 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 Application

Using the unparser to emit code is mostly straightforward. The New Jersey Machine-Code Toolkit (Ramsey and Fernández 1995) uses the unparser to emit C code. Most of the expressions in the C code come from solving equations (Ramsey 1996). Internally, the toolkit represents expressions using a different ML constructor for each operator; this representation facilitates algebraic simplification. Emitting C code therefore takes three steps: transforming the internal representation of expressions to the simpler representation used by the unparser, running the unparser to get input for a prettyprinter, and finally running the prettyprinter to get C source code. The prettyprinter uses a model like that of Pugh and Sinofsky (1987), but it uses a dynamic-programming algorithm to convert to text.

Assigning precedence and associativity is 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`. 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. These functions are used in turn to define functions that create values of type `Unparser.ast`, e.g.,

```
fun binary $ =
  let val optext = pptext (" " ^ $ ^ " ")
  in  fn (l, r) => Unparser.BINARY(l, (optext, Cprec $, Cfixity $), r)
  end
val />>/ = binary ">>"
 ⋮
```

where `pptext` turns a string into input for the prettyprinter.

Functions like `/>>/` can be declared infix in ML; by careful use of such declarations, we can generate C code by using ML code that is reminiscent of C code. For example, this code fragment transforms the toolkit's internal representation of a range test into the `Unparser.ast` representation of the ultimate C code. The internal form `U.INRANGE(e, {lo, hi})` represents the predicate $\texttt{lo} \leq \texttt{e} < \texttt{hi}$, in which `e` is an arbitrary expression, and `lo` and `hi` are integers known at transformation time.

⟨*parts of transformation to C*⟩≡
```
  fun exp(U.INRANGE(e, {lo, hi})) =
        if lo + 1 = hi then exp e /==/ ppcon lo
        else ppcon lo /<=/ exp e /&&/ exp e /</ ppcon hi
```

where `ppcon` converts an integer to a fragment of prettyprinter input, which is of the proper type `Atom.atom`.

The fixity of operators is not always exactly what one might expect. 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 never expressions but identifiers, and semantically they are more like unary operators. The unparser in fact treats them as unary postfix operators, choosing the representation dynamically according to what member is being selected. For example, we write

⟨*parts of transformation to C*⟩+≡
```
    | exp(U.SELECT(e, membername)) = exp e dot membername
```

where `dot` dynamically builds a postfix operator:

```
fun dot (e, tag) =
  Unparser.UNARY((pp.te ("." ^ tag), Cprec ".", Assoc.POSTFIX), e)
infix 9 dot
```

Some fragments of C may be unparsed before others. For example, in translating array access, the subscript appears in square brackets, so it can be unparsed independently. The subscript operation itself is represented by the empty string, but it is given its proper precedence and associativity, which, for example, enable the unparser to produce either `*p[n]` or `(*p)[n]` as required.

⟨*parts of transformation to C*⟩+≡
```
    | exp(U.ARRAY_SUB(a, n)) =
        let val index = pplist (unparse (exp n))
            val subscript = pplist [pptext "[", index, pptext "]"]
        in  Unparser.BINARY(exp a,
                            (pptext "", Cprec "subscript", L),
                            Unparser.AST_ATOM subscript)
        end
```

A similar trick is used to unparse function calls after applying an n-ary comma operator to the arguments. The function `pplist` concatenates a list of prettyprinter atoms; it is also used to define the function `parenthesize` that is passed to the unparser:

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

Putting the unparser together with a simple prettyprinter can produce readable C code. Here is code that emits a Pentium instruction to call a procedure identified by an indexed addressing mode:

```
  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");
```

Without an unparser based on precedence and associativity, we would have extra parentheses. Parenthesizing every expression is too awful to contemplate, but we can see what happens if we parenthesize all non-atomic expressions:

```
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"));
```

This code is ugly, but manageable. Code with more complicated expressions quickly becomes unreadable. Luckily, just is it has been plugged into a rewritten machine-code toolkit, the unparser in this paper can be plugged into any other tool that needs to emit idiomatic code in a high-level language.

References

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. 1988.
The AWK Programming Language.
Reading, MA: Addison-Wesley.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley.

Hughes, John. 1995.
The Design of a Pretty-printing Library.
In Jeuring, J. and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925.

Knuth, Donald E. 1984.
Literate programming.
The Computer Journal, 27(2):97–111.

Milner, Robin, Mads Tofte, and Robert W. Harper. 1990.
The Definition of Standard ML.
Cambridge, Massachusetts: MIT Press.

Oppen, Derek C. 1980 (October).
Prettyprinting.
ACM Transacations on Programming Languages and Systems, 2(4):465–483.

Paulson, Laurence C. 1991.
ML for the working programmer.
New York: Cambridge University Press.

Pugh, William W. and Steven J. Sinofsky. 1987 (January).
A new language-independent prettyprinting algorithm.
Technical Report TR 87-808, Cornell University.

Ramsey, Norman and Mary F. Fernández. 1995 (January).
The New Jersey Machine-Code Toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

Ramsey, Norman. 1996 (April).
A simple solver for linear equations containing nonlinear operators.
Software—Practice & Experience, 26(4):467–487.

Ullman, Jeffrey D. 1994.
Elements of ML Programming.
Englewood Cliffs, NJ: Prentice Hall.

Wirth, Niklaus. 1977 (November).
What can we do about the unnecessary diversity of notation for syntactic definitions?
Communications of the ACM, 20(11):822–823.

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, and it automatically inserts `juxtarator` whenever two expressions are juxtaposed in the input. 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.

We encode these invariants in the type system only in part—the type system does not actually distinguish prefix and infix operators, so the names `prefix` and `infixop`[1] are abbreviations only.

⟨*general*⟩+≡
```
type infixop = rator
type prefix = rator
datatype stack = BOT
               | BIN of infixop * ast * prefix list * stack
type stacktop = (prefix list * ast)
```

Again, we create the bogus operator `minrator` to appear on the bottom of the stack:

⟨*general*⟩+≡
```
val minrator = (Atom.bogus, Atom.minprec, INFIX NONASSOC)
fun srator  (BIN ($, _, _, _)) = $
  | srator  BOT = minrator
```

The input we're parsing looks something like this:

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

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 for the reader. This function identifies prefix operators.

⟨*general*⟩+≡
```
fun isPrefix (_, _, f) = f = PREFIX : bool
```

[1] `infix` is a reserved word in ML.

We use two parsing functions, `parse_prefix` and `parse_postfix`, depending on whether we are before or after an atomic input. The overall structure of the parser is

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

There are many cases. `parse_prefix` saves prefix operators until reaching 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 `parse_prefix` encounters a non-prefix operator or end of file, the input is not correct. When `parse_prefix` encounters an atomic input, it passes input and the saved prefixes to `parse_postfix`.

⟨*general parsing functions*⟩≡
```
fun parse_prefix(stack, prefixes, RATOR $ :: ipts') =
        if isPrefix $ then
          parse_prefix(stack, $ :: prefixes, ipts')
      else
        raise ParseError("%s is not a prefix operator", [$])
  | parse_prefix(_, _, []) = raise ParseError ("premature EOF", [])
  | parse_prefix(stack, prefixes, EXP a :: ipts') =
        parse_postfix(stack, (parse_atom a, prefixes), ipts')
```

In addition to the stack and the input, `parse_postfix` maintains a current expression `e`, and a list `prefixes` containing unreduced prefix operators that immediately precede `e`. The general 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*⟩+≡

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

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

⟨*general parsing functions*⟩+≡

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

The insertion itself is straightforward.

⟨*insert* `juxtarator`⟩≡

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

The other major case for `parse_postfix` 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*⟩+≡

```
  | parse_postfix(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 ($, e', prefixes, stack') =>
                       parse_postfix(stack', (BINARY(e', $, e), prefixes), ipts)
           | BOT => raise Impossible (*  BOT has lowest precedence *)
      else if noparens(irator, srator stack, RIGHT) then
        case ifixity
          of POSTFIX => (* reduce postfix rator *)
               parse_postfix(stack, (UNARY(irator, e), []), ipts')
           | INFIX _ => (* shift, look for prefixes *)
               parse_prefix(BIN(irator, e, [], stack), [], ipts')
           | PREFIX  => ⟨insert juxtarator⟩
      else
         raise ParseError ("%s is non-associative", [irator])
```

When the input is exhausted, we reduce the prefixes, then the stack, until we finally have the result.

⟨*general parsing functions*⟩+≡

```
  | parse_postfix(stack, (e, $ :: prefixes), []) = (* reduce prefix *)
      parse_postfix(stack, (UNARY($, e), prefixes), [])
  | parse_postfix(BIN ($, e', prefixes, stack'), (e, []), []) = (* reduce stack *)
      parse_postfix(stack', (BINARY(e', $, e), prefixes), [])
  | parse_postfix(BOT, (e, []), []) = e
```

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

⟨*general parsing functions*⟩+≡

```
  and parse(IMAGE(l)) = parse_prefix(BOT, [], l)
  and parse_atom (LEX_ATOM a) = AST_ATOM a
    | parse_atom (PARENS im)   = parse im
```