Parsing Mixfix Operators

Nils Anders Danielsson[1],* and Ulf Norell[2]

[1] University of Nottingham
[2] Chalmers University of Technology

Abstract. A simple grammar scheme for expressions containing mixfix operators is presented. The scheme is parameterised by a precedence relation which is only restricted to be a directed acyclic graph; this makes it possible to build up precedence relations in a modular way. Efficient and simple implementations of parsers for languages with user-defined mixfix operators, based on the grammar scheme, are also discussed. In the future we plan to replace the support for mixfix operators in the language Agda with a grammar scheme and an implementation based on this work.

1 Introduction

Programming language support for user-defined infix operators is often nice to have. It enables the use of compact and/or domain-specific notations, especially if a character set with many symbols is used. The feature can certainly be abused to create code which the intended audience finds hard to read, but the inclusion of user-defined infix operators in a number of programming languages, including Haskell (Peyton Jones 2003), ML (Milner et al. 1997), Prolog (Sterling and Shapiro 1994), and Scala (Odersky 2009), suggests that this is a risk which many programmers are willing to take.

Some languages, such as Coq (Coq Development Team 2009), Isabelle (Paulson et al. 2008), and Obj (Goguen et al. 1999), take things a step further by supporting the more general concept of *mixfix* (also known as *distfix*) operators. A mixfix operator can have several name parts and be infix (like the typing relation $\_\vdash\_:\_$), prefix (`if_then_else_`), postfix (array subscripting: `_[_]`), or closed (Oxford brackets: $[\![\_]\!]$). With mixfix operators the advantages of binary infix operators can be taken one step further, but perhaps also the disadvantages.

An important criterion when designing a programming language feature is that the feature should be easy to understand. In the case of mixfix operators the principle explaining how to parse an arbitrary expression should be simple (even though abuse of the feature may lead to a laborious parsing process). Mixfix operators are sometimes perceived as being difficult in this respect. Our aim with this work is to present a method for handling mixfix operators which is elegant, easy to understand, and easy to implement with sufficient efficiency.

We show how to construct a simple expression grammar, given a set of operators with specified precedence and associativity (see Sect. 3). We want to avoid

* The author would like to thank EPSRC for financial support.

monolithic precedence relations in which every operator is related to every other, so we only require the precedences to form a directed acyclic graph (Sect. 2). To abstract from source language details the language mechanisms used to specify the operators, precedence graphs etc. are left unspecified.

In Sect. 5 the grammars are defined formally, using the total, dependently typed language Agda (Norell 2007; Agda Team 2009). The grammars are defined using parser combinators with a well-defined semantics, so as a side effect a provably correct implementation of a mixfix expression parser is obtained. The formalisation also shows that the expressions generated by the grammars correspond exactly to a notion of precedence correct expressions. Furthermore the Agda code provides a number of examples of the use of mixfix operators.

The restriction of precedence relations to *acyclic* graphs ensures that the constructed grammars are unambiguous, assuming that all operator name parts are unique (Sect. 4). Acyclicity also ensures that parsers corresponding to the generated grammars can be implemented using (backtracking) recursive descent. However, naive use of recursive descent leads to poor performance. In a prototype which uses *memoising* parser combinators (Frost and Szydlowski 1996) we have achieved sufficient efficiency along with a simple implementation (Sect. 6). In the future we plan to use this approach to handle mixfix operators in Agda.

The paper ends with a discussion of related work and some conclusions in Sects. 7–8.

2 Precedence and Associativity

In some languages it is very easy to introduce mixfix operators. For instance, in Agda a name is an operator if it includes an underscore character. Every underscore stands for an operator *hole*, i.e. a position in which an argument expression is expected. The examples from Sect. 1 are valid operator names in Agda:[3] `_⊢_:_`, `_[_]`, `if_then_else_`, and `⟦_⟧`. However, typically one wants to combine operators into larger expressions, and then it is important to specify if and how they can be combined. How should an expression like

$$\texttt{if b} \wedge \texttt{n + n == n ! then n else (n + n - n)} \tag{1}$$

be parsed, for instance?

The traditional way to disambiguate such expressions is to use *precedence* and *associativity* (Aho et al. 1986; Aasa 1995), and we follow this approach. Precedence specifies if an operator "binds tighter" than another. Associativity specifies how sequences of infix operators of the same precedence should be interpreted; an operator can be left associative, right associative, or non-associative. The expression `x + y * z` parses as `x + (y * z)` if `_*_` binds tighter than `_+_`, and `x + y - z` parses as `(x + y) - z` if the two operators have the same precedence and are both left associative. See Table 1 for a summary of how precedence and associativity affect parsing of infix operators.

[3] Unless the wrong colon character is used.

Table 1. Possible outcomes of parsing `x + y * z`, where `_+_` and `_*_` are infix operators. Here + < * means that `_*_` binds tighter than `_+_`, and + = * means that the operators have equal precedence.

Precedence	Associativity	Result of parsing `x + y * z`
+ < *		x + (y * z)
* < +		(x + y) * z
+ = *	Both left	(x + y) * z
+ = *	Both right	x + (y * z)
None of the above		Parse error

Unlike languages like Coq and Isabelle, but following Aasa (1995), we do not assign any form of precedence to the internal holes of mixfix operators (i.e. the holes which are surrounded by name parts). Instead, to keep things simple, expressions of arbitrary precedence are allowed in the holes. This means that one can effectively define parentheses as a closed mixfix operator `(_)`, binding tighter than everything else, with the semantics of the polymorphic identity function (as long as the name parts `(` and `)` are unambiguous).

Many languages require the precedence relation to be a total order. However, this can make reading source code more difficult, because it means that every operator is related to every other, which is likely to make it harder for programmers to remember the precedence relation. It also means that one needs to make unnecessary, possibly arbitrary choices. Why should one have to specify a relation between `_+_` and _∧_, two semantically unrelated operators, for instance? This goes against modularity.

One might think that partial orders are a good alternative. However, under the (reasonable) assumption that `_+_` binds tighter than `_==_`, which binds tighter than _∧_, transitivity would imply that `_+_` binds tighter than _∧_, which we want to avoid.

Instead we just require that the precedence relation forms a directed acyclic graph (DAG), where an edge from one node to another means that the operators in the second node bind tighter than those in the first one, and operators in the same node have equal precedence. This makes it possible to define a small domain-specific library (language) with a couple of operators and a natural, possibly domain-specific precedence relation, without relating these operators to those from other libraries. However, we note that partial and total orders are DAGs, so the results below apply also to those cases.

We require the graphs to be acyclic because cyclic precedence relations very easily lead to ambiguous grammars. Furthermore acyclicity ensures that the grammars are not left recursive, thus enabling backtracking recursive descent as an implementation technique.

Figure 1 contains a small precedence graph. To keep things simple the grammar scheme introduced below is concerned solely with operators, so the precedence graph includes some variables (`b` and `n`) and parentheses (`(_)`). These are treated as closed mixfix operators which bind more tightly than all other

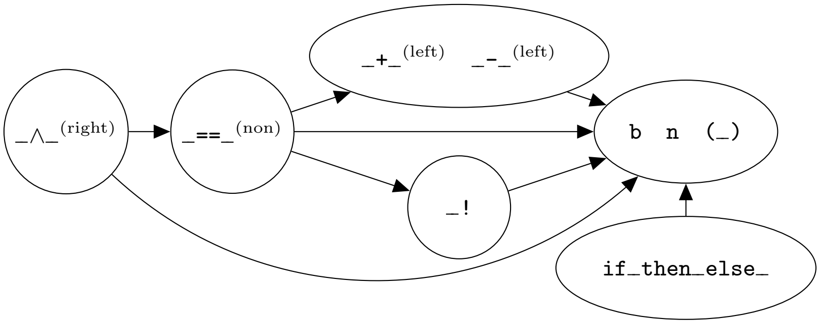

Fig. 1. A precedence graph. An arrow from node *p* to node *q* means that operators in node *q* bind tighter than operators in node *p*. Infix operators are annotated with their associativity.

operators. Let us interpret the example (1) given above using this precedence graph.

We can start by noticing that all the operators' name parts are unique, so it is easy to identify which name parts belong to which operator. We can then isolate the internal arguments of `if_then_else_` and `(_)`:

$$\texttt{if}\ [\texttt{b} \wedge \texttt{n + n == n !}]\ \texttt{then}\ [\texttt{n}]\ \texttt{else (}\ [\texttt{n + n - n}]\ \texttt{).} \tag{2}$$

Furthermore parentheses bind more tightly than `if_then_else_`, so the parenthesised expression has to be the last argument to the conditional:

$$\texttt{if}\ [\texttt{b} \wedge \texttt{n + n == n !}]\ \texttt{then}\ [\texttt{n}]\ \texttt{else}\ \big[\texttt{(}\ [\texttt{n + n - n}]\ \texttt{)}\big]. \tag{3}$$

In `b ∧ n + n == n !` the operator _∧_ is only related to `_==_` (plus the variables), and it binds weaker, so the only possible parse is

$$[\texttt{b}]\ \wedge\ \Big[\big[[\texttt{n}]\ \texttt{+}\ [\texttt{n}]\big]\ \texttt{==}\ \big[[\texttt{n}]\ \texttt{!}\big]\Big]. \tag{4}$$

Note that when reading source code which is known to be syntactically correct (which is the case here) "the only possible parse" translates into "the one and only correct parse". Note also that the interpretation of this subexpression would have been slightly less straightforward if the precedence relation had been a partial order.

Finally the operators `_+_` and `_-_` have the same precedence and are both left associative, so we end up with

$$\begin{gathered}\texttt{if}\ \Big[[\texttt{b}]\ \wedge\ \Big[\big[[\texttt{n}]\ \texttt{+}\ [\texttt{n}]\big]\ \texttt{==}\ \big[[\texttt{n}]\ \texttt{!}\big]\Big]\Big] \\ \texttt{then}\ [\texttt{n}]\ \texttt{else}\ \Big[\texttt{(}\ \Big[\big[[\texttt{n}]\ \texttt{+}\ [\texttt{n}]\big]\ \texttt{-}\ [\texttt{n}]\Big]\ \texttt{)}\Big].\end{gathered} \tag{5}$$

3 A Grammar Scheme for Mixfix Operators

Section 2 may have given some intuition about precedence and associativity, but there are still some design choices left. This section makes things precise by giving a grammar scheme which, when instantiated with a precedence graph, yields a context-free grammar specifying the syntax of expressions.

First some definitions:

- A mixfix operator consists of a finite sequence of holes (denoted by _ above) and name parts (`if`, `then` and `else` in `if_then_else_`), plus in the case of an infix operator an associativity (left, right or "non"; note that we do not restrict the term "infix operator" to binary operators). To reduce the risk of ambiguity we require that operators contain at least one name part, and that two holes may not occur next to each other in an operator. Furthermore, for simplicity, we require that two name parts may not occur next to each other either.
- A precedence graph is a finite directed acyclic simple graph with unlabelled edges, whose nodes are annotated with finite sets of operators.

Given such a precedence graph a grammar is constructed. The terminals of the grammar are the name parts used by the operators in the graph.

We make no assumptions about uniqueness of name parts or operators (except that a given operator may only occur once in a given node). The resulting grammar can hence be ambiguous. We feel that it is overly restrictive to require the grammar to be unambiguous. For instance, it seems unnecessary to reject a program just because two imported libraries both define a particular operator, even though this operator is never used, or only used in such a way that it can be disambiguated based on context. As another example, the designer of some library may want to include both `if_then_` and `if_then_else_`, and in order to keep the library simple it seems reasonable to give both operators the same precedence. Given the rules below this makes the grammar ambiguous, because `if e then if e then e else e` can be parsed in two ways.[4] Instead of rejecting ambiguous grammars we suggest that ambiguous *parses* should be rejected, preferably together with error messages showing all possible parses, thus aiding debugging. Language designers are of course free to impose further restrictions to ensure unambiguity. (Ambiguity is discussed further in Sect. 4.)

As mentioned above the language of the constructed grammar *only* contains operator applications (possibly nullary). Expressions in a real language usually contain other constructions as well, like parentheses, let or lambda expressions and non-operator symbols. To keep things simple such constructs are not treated here, but we note that it is easy to incorporate several of them by modifying the grammar scheme (see Sect. 5.4 for one example).

Let us now build up the grammar scheme step by step, starting with a precedence graph where every node is labelled with exactly one infix, non-associative

[4] Assuming that `e` binds tighter than the conditionals.

operator with no internal holes. In this case an expression headed by the operator *op* from graph node p consists of an expression headed by an operator which binds tighter, then *op*'s only name part, and finally another expression headed by an operator which binds tighter. We can encode this using the non-terminals

$$\widehat{p} ::= p\!\uparrow \ op_p^{\text{non}} \ p\!\uparrow \quad \text{and} \tag{6}$$

$$p\!\uparrow ::= \bigvee \{\, \widehat{q} \mid p < q \,\}\,. \tag{7}$$

Here $\bigvee S$ stands for a choice between all the elements in the finite set S, op_p^{non} is the single name part of the (non-associative) operator in node p, and $p < q$ means that there is an edge from node p to node q. An arbitrary expression is an expression headed by an arbitrary operator, so the non-terminal for expressions is

$$\mathit{expr} ::= \bigvee \{\, \widehat{p} \mid p \text{ is a graph node} \,\}\,. \tag{8}$$

It is straightforward to extend this grammar scheme to infix, non-associative operators with multiple name parts. All the internal holes can contain arbitrary expressions, so we can just let op_p^{non} stand for the non-terminal

$$op_p^{\text{non}} ::= n_1 \ \mathit{expr} \ n_2 \ \mathit{expr} \ \cdots \ n_k, \tag{9}$$

where n_i is the i-th name part of the non-associative infix operator with k name parts annotating node p. For graphs whose nodes are annotated with sets of operators we change the definition of op_p^{non} to include one production for every operator in node p.

Finally let us include other kinds of operators. This amounts to adding more productions to $\widehat{p}$. When is an expression headed by a right associative infix operator precedence correct? Both arguments should be allowed to be expressions headed by operators which bind tighter, and the right argument should also be allowed to be an application of another right associative operator of the same precedence. There is scope for allowing other combinations to be precedence correct as well, though. We choose to view prefix operators as right associative by including the productions

$$\widehat{p} ::= \overrightarrow{p}^{+} \ p\!\uparrow \quad \text{and} \tag{10}$$

$$\overrightarrow{p} ::= op_p^{\text{prefix}} \mid p\!\uparrow \ \ op_p^{\text{right}}. \tag{11}$$

Here e^{+} stands for a positive number of repetitions of e, and op_p^{prefix} and op_p^{right} are the analogues of op_p^{non} for prefix and right associative operators, respectively. (In Sect. 7 a grammar scheme due to Aasa (1995) which handles prefix operators differently is discussed.)

Note that the parse trees generated for prefix and right associative operators are not the correct ones; for prefix operators they have the shape $(op \cdots op)\mathit{rest}$ rather than $op(\cdots(op \ \mathit{rest})\cdots)$. However, this is easily corrected by a post-processing pass. In our implementation based on parser combinators one simply

$$
\begin{aligned}
\mathit{expr} &::= \bigvee \{\, \widehat{p} \mid p \text{ is a graph node} \,\} \\
\widehat{p} &::= \mathit{op}_p^{\text{closed}} \\
&\quad\mid p{\uparrow}\ \mathit{op}_p^{\text{non}}\ p{\uparrow} \\
&\quad\mid \overrightarrow{p}^{+}\ p{\uparrow} \\
&\quad\mid p{\uparrow}\ \ \overleftarrow{p}^{+} \\
\overrightarrow{p} &::= \mathit{op}_p^{\text{prefix}} \mid p{\uparrow}\ \ \mathit{op}_p^{\text{right}} \\
\overleftarrow{p} &::= \mathit{op}_p^{\text{postfix}} \mid \mathit{op}_p^{\text{left}}\ p{\uparrow} \\
p{\uparrow} &::= \bigvee \{\, \widehat{q} \mid p < q \,\} \\
\mathit{op}_p^{\mathit{fix}} &::= \bigvee \left\{\ n_1\ \ \mathit{expr}\ n_2\ \ \mathit{expr}\ \cdots\ n_k \ \middle|\ \begin{array}{l} n_1, \ldots, n_k \text{ are the name parts of} \\ \text{an operator in node } p \text{ with fix-} \\ \text{ity/associativity } \mathit{fix} \end{array} \right\}
\end{aligned}
$$

Fig. 2. A grammar scheme for mixfix expressions, parameterised by a precedence graph.

$$
\begin{aligned}
\widehat{p} &::= \widehat{p}^{\text{closed}} \mid \widehat{p}^{\text{non}} \mid \widehat{p}^{\text{right}} \mid \widehat{p}^{\text{left}} \\
\widehat{p}^{\text{closed}} &::= \mathit{op}_p^{\text{closed}} \\
\widehat{p}^{\text{non}} &::= p{\uparrow}\ \mathit{op}_p^{\text{non}}\ p{\uparrow} \\
\widehat{p}^{\text{right}} &::= \left(\mathit{op}_p^{\text{prefix}} \mid p{\uparrow}\ \mathit{op}_p^{\text{right}}\right)\ \left(\widehat{p}^{\text{right}} \mid p{\uparrow}\right) \\
\widehat{p}^{\text{left}} &::= \left(\widehat{p}^{\text{left}} \mid p{\uparrow}\right)\ \left(\mathit{op}_p^{\text{postfix}} \mid \mathit{op}_p^{\text{left}}\ p{\uparrow}\right)
\end{aligned}
$$

Fig. 3. An alternative formulation of $\widehat{p}$ which leads to grammars which are left and right recursive, but whose parse trees do not require post-processing.

needs to include a right fold in the semantic action attached to the $\widehat{p}$ production (see Sect. 5). Furthermore this formulation has the advantage of not being right recursive.

Note also that it would be reasonable to allow prefix operators to be non-associative. For instance, if the operators `if_then_` and `if_then_else_` from the example above were both non-associative then `if e then if e then e else e` could only be parsed in one way:[5] `if e then (if e then e) else e`. To keep things simple we treat all prefix operators as right associative in this presentation, though.

Postfix operators, left associative infix operators and closed operators can be handled analogously. The full grammar scheme is shown in Fig. 2. Note that, because precedence graphs are acyclic, the instantiated grammars are neither left nor right recursive (see Sects. 5.4–5.5). An alternative definition of $\widehat{p}$ which avoids the need to post-process the parse trees and accepts the same strings is

[5] Unless the grammar contains some other ambiguity.

given in Fig. 3. The grammars resulting from this definition can be left and right recursive, though.

To make things more concrete, let us instantiate the grammar scheme for the precedence graph in Fig. 1. After some simplification—removal of productions which are unused or always fail, together with inlining—we get the following grammar (with terminals underlined):

$$\begin{aligned}
\mathit{expr} &::= \mathit{and} \mid \mathit{eq} \mid \mathit{term} \mid \mathit{fac} \mid \mathit{if} \mid \mathit{closed} \\
\mathit{and} &::= (\mathit{and}{\uparrow}\ \ \underline{\triangle})^{+}\ \mathit{and}{\uparrow} \\
\mathit{and}{\uparrow} &::= \mathit{eq} \mid \mathit{closed} \\
\mathit{eq} &::= \mathit{eq}{\uparrow}\ \ \underline{\texttt{==}}\ \ \mathit{eq}{\uparrow} \\
\mathit{eq}{\uparrow} &::= \mathit{term} \mid \mathit{fac} \mid \mathit{closed} \\
\mathit{term} &::= \mathit{closed}\ \ ((\underline{\texttt{+}} \mid \underline{\texttt{-}})\ \mathit{closed})^{+} \\
\mathit{fac} &::= \mathit{closed}\ \ \underline{\texttt{!}}^{+} \\
\mathit{if} &::= (\underline{\texttt{if}}\ \mathit{expr}\ \underline{\texttt{then}}\ \mathit{expr}\ \underline{\texttt{else}})^{+}\ \mathit{closed} \\
\mathit{closed} &::= \underline{\texttt{b}} \mid \underline{\texttt{n}} \mid \underline{\texttt{(}}\ \mathit{expr}\ \underline{\texttt{)}}
\end{aligned} \tag{12}$$

4 Unambiguity

An important property of the grammar scheme introduced in Sect. 3 is that, while the instantiated grammars can in general be ambiguous, this ambiguity is necessarily introduced by reusing the same name part in several operators: if all operator name parts in a precedence graph are unique, then the resulting grammar is unambiguous.

If no operators have internal holes then this result can be proved by adapting a theorem due to Lotfallah (2009). The general case can be reduced to the simpler one by using the following observation (following Aasa (1995)): Operators with internal name parts act like generalised brackets, so given a precedence graph with unique name parts and a syntactically correct string one can uniquely identify the substrings corresponding to the non-terminals op_p^{fix}. One can then treat every instance of one of the op_p^{fix} non-terminals as a terminal of a new grammar. For instance, the string `if b ∧ b else n then n` would be treated as a string containing two terminals: `if b ∧ b else n then` and `n`. This new grammar only contains atoms, unary prefix and postfix operators and binary infix operators, so unambiguity follows from Lotfallah's theorem. It remains to show that the internal expressions of the op_p^{fix} "terminals" are unambiguous, but this follows by applying the same argument inductively to the proper substrings corresponding to the *expr* non-terminal.

Note that Lotfallah's theorem requires a form of acyclicity. Cyclic precedence graphs easily lead to ambiguous grammars, even if all name parts are unique. Consider the graph in Fig. 4, for instance. The grammar corresponding to this graph can generate the string `0 * 0 + 0` in two ways: with `*` as the outermost operator (`0 * (0 + 0)`) or with `+` as the outermost operator (`(0 * 0) + 0`).

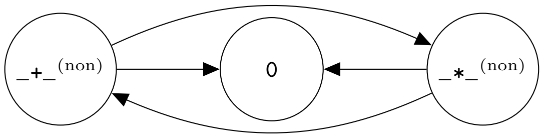

Fig. 4. A cyclic precedence graph.

5 Formalisation

The unambiguity result from Sect. 4 indicates that the grammars are, in some sense, useful: reasonable expressions will not be rejected because of ambiguities, as long as all name parts are unique. In this section a notion of precedence correct expression is defined, and it is proved that the grammars generate exactly the precedence correct expressions. This amounts to a different aspect of usability: no precedence correct expression will be rejected because the grammar is too limited.

We perform this exercise formally, in the total, dependently typed functional programming language Agda (Agda Team 2009). As part of this development the grammar scheme introduced above is defined formally using parser combinators. First operators, precedence graphs and precedence correct expressions are defined (Sects. 5.1–5.3), then a parser combinator library is introduced (Sect. 5.4), the grammar scheme is defined (Sect. 5.5), and finally the proof mentioned above is outlined (Sect. 5.6). Note that some minor details of Agda have been changed in order to aid the presentation.

5.1 Operators

Before defining what an operator is we encode associativities and fixities using two simple data types. Fixities are combined with associativities, but only for infix operators; prefix, postfix and closed operators are viewed as being right, left and non-associative, respectively:

```
data Associativity : Set where
  left   : Associativity
  right  : Associativity
  non    : Associativity
data Fixity : Set where
  prefx   : Fixity
  infx    : Associativity → Fixity
  postfx  : Fixity
  closed  : Fixity
```

(The constructors of a data type are introduced by giving their type signatures. Note that **infix** is a reserved word in Agda, hence the strange names.)

An operator is then represented by its fixity plus a vector containing its name parts. Note that the fixity is exposed in the type, along with the internal arity, i.e. the number of internal arguments:

```
record Operator (fix : Fixity) (arity : ℕ) : Set where
  field nameParts : Vec NamePart (1 + arity)
```

(*Vec A n* is a list of *A*s of length *n*. *NamePart* is the type of name parts.) The operator `if_then_else_` is represented as follows:

```
if-then-else : Operator prefx 2
if-then-else = record {nameParts = "if" :: "then" :: "else" :: []}
```

5.2 Precedence Graphs

For simplicity precedence graphs are represented by their unfoldings as forests, with one tree (*Precedence*) for every node in the graph:

```
data Precedence : Set where
  precedence : ((fix : Fixity) → List (∃ (Operator fix))) →
               List Precedence → Precedence

PrecedenceGraph : Set
PrecedenceGraph = List Precedence
```

Two projection functions are defined for the *Precedence* nodes, one returning the operators of the given precedence, and one returning the successor nodes:

```
ops : Precedence → (fix : Fixity) → List (∃ (Operator fix))
ops (precedence o s) = o

↑ : Precedence → List Precedence
↑ (precedence o s) = s
```

(Dependent function spaces are written as $(x : A) \to B$.) Note that the *set* of operators annotating a graph node is represented by a function mapping a fixity to a *list* of operators of that fixity; the stronger invariants of a set are not needed for this development.

The type ∃ is used to hide the arity argument of *Operator fix*, so that operators of different arity can be members of the same list. It is a variant of the pair type:

```
data ∃ {A : Set} (B : A → Set) : Set where
  _,_ : (x : A) → B x → ∃ B
```

(Note that arguments in braces, like $\{A\}$, are *implicit*; they do not need to be given explicitly if Agda can infer them.)

5.3 Expressions

The type of expressions which are precedence correct with respect to a given precedence graph is defined in a module parameterised by the graph:

```
module PrecedenceCorrect (g : PrecedenceGraph) where
```

The definition consists of four mutually inductive types:

– *Expr ps* stands for expressions where the head operator has one of the precedences in *ps*:

```
data Expr (ps : List Precedence) : Set where
  _•_ : ∀ {p assoc} → p ∈ ps → Ex p assoc → Expr ps
```

  Ex p assoc, introduced below, stands for expressions where the head operator has precedence *p* and associativity *assoc*. The type _∈_ encodes list membership:

```
data _∈_ {A : Set} : A → List A → Set where
  here  : ∀ {x    xs}          → x ∈ x :: xs
  there : ∀ {x y xs} → x ∈ xs → x ∈ y :: xs
```

– *In ops* stands for the application of one of the operators in *ops* to all its internal arguments:

```
data In {fix} (ops : List (∃ (Operator fix))) : Set where
  _•_ : ∀ {arity op} →
          (arity, op) ∈ ops → Vec (Expr g) arity → In ops
```

  Note that the internal arguments are unrestricted expressions (*Expr g*). Note also that constructors are overloaded in Agda.

– *Out p assoc* contains expressions where the head operator either has precedence *p* and associativity *assoc*, or binds strictly tighter than *p*:

```
data Out (p : Precedence) (assoc : Associativity) : Set where
  similar : Ex p assoc → Out p assoc
  tighter : Expr (↑ p) → Out p assoc
```

  Out p left stands for the left arguments of left associative operators of precedence *p*, and similarly for *Out p* right.

– Finally *Ex p assoc* is defined. Note the use of mixfix operators:

```
data Ex (p : Precedence) : Associativity → Set where
  ⟪_⟫    :                In (ops p closed      )                → Ex p non
  _⟨_⟫   : Out p left  → In (ops p postfx      )                → Ex p left
  ⟪_⟩_   :                In (ops p prefx       ) → Out p right → Ex p right
  _⟨_⟩_  : Expr (↑ p) → In (ops p (infx non)  ) → Expr (↑ p)  → Ex p non
  _⟨_⟩^ℓ_ : Out p left  → In (ops p (infx left) ) → Expr (↑ p)  → Ex p left
  _⟨_⟩^r_ : Expr (↑ p) → In (ops p (infx right)) → Out p right → Ex p right
```

Two "weakening" functions will also be used. The function *weakenE* takes an expression headed by an operator which has one of the precedences in *ps* and converts it to an expression headed by an operator with one of the precedences in $p :: ps$, and *weakenI* is similar:

```
weakenE : ∀ {p ps} → Expr ps → Expr (p :: ps)
weakenE (p∈ps • e) = there p∈ps • e
weakenI : ∀ {fix ops} {op : ∃ (Operator fix)} → In ops → In (op :: ops)
weakenI (op∈ops • args) = there op∈ops • args
```

5.4 Parser Combinators

In order to define the grammar scheme we will use a parser combinator library based on that described by Danielsson and Altenkirch (2009), but tailored specifically for this task. The type *Parser* defines the parser combinators:

```
data Parser  : Set → Set where
  fail       : ∀ {A} → Parser A
  _|_        : ∀ {A} → Parser A → Parser A → Parser A
  _∥_        : ∀ {I i} {A : I → Set} →
               Parser (A i) → Parser (∃ A) → Parser (∃ A)
  _⊛_        : ∀ {A B} → Parser (A → B) → Parser A → Parser B
  _<$>_      : ∀ {A B} → (A → B) → Parser A → Parser B
  _+         : ∀ {A} → Parser A → Parser (List⁺ A)
  _between_  : ∀ {A n} → ∞ (Parser A) → Vec NamePart (1 + n) →
               Parser (Vec A n)
```

The semantics of the parser combinators is given by the following inductively defined type:

```
data _∈_·_ : ∀ {A} → A → Parser A → List NamePart → Set where
  . . .
```

The type $x \in p \cdot s$ is inhabited if and only if one of the possible results of applying the parser p to the string s is x. The parser combinators come with a parser backend which takes a parser and a string and computes all parses matching the string. Because Agda is total (modulo any bugs in the implementation) this backend is guaranteed to terminate, and it has been proved to be sound and complete with respect to the semantics.

Let us now explain all the combinators. The parser fail always fails, so there is no constructor for it in _∈_·_. The combinator _|_ encodes symmetric choice:

$$
\begin{aligned}
|^{\ell} &: x \in p_1 \cdot s \to x \in p_1 \mid p_2 \cdot s \\
|^{r} &: x \in p_2 \cdot s \to x \in p_1 \mid p_2 \cdot s
\end{aligned}
$$

(The introduction of the bound variables x, s, p_1 and p_2 has been omitted here to avoid clutter.) The combinator _∥_ is a variant of _|_:

$$\begin{array}{lll}
\|^{\ell} & : x \in p_1 \;\cdot\; s \to (\_\;,\; x) & \in p_1 \parallel p_2 \;\cdot\; s \\
\|^{r} & : x \in p_2 \;\cdot\; s \to x & \in p_1 \parallel p_2 \;\cdot\; s
\end{array}$$

(The underscore tells Agda to try to infer what the corresponding expression should be.) The $\_\circledast\_$ operator is *applicative functor application* (McBride and Paterson 2008): the result of $p_1 \circledast p_2$ is the result of p_1 (a function) applied to the result of p_2. The combinator $\_{<}\${>}\_$ maps a function over the results of a parser:

$$\begin{array}{lll}
\_\circledast\_ & : f \in p_1 \;\cdot\; s_1 \to & x \in p_2 \;\cdot\; s_2 \to f\; x \in p_1 \circledast p_2 \;\cdot\; s_1 +\!\!+\; s_2 \\
{<}\${>}\_ & : & x \in p \;\;\cdot\; s \;\;\to f\; x \in f\; {<}\${>}\; p \;\cdot\; s
\end{array}$$

(The function $\_{+\!\!+}\_$ concatenates two lists.) The parser $p\,+$ parses one or more occurrences of p:

$$\begin{array}{ll}
+_{[\,]} & : x \in p \;\cdot\; s \qquad\qquad\qquad\qquad\;\; \to [x] \qquad \in p + \;\cdot\; s \\
+_{::} & : x \in p \;\cdot\; s_1 \to xs \in p + \;\cdot\; s_2 \to x :: xs \in p + \;\cdot\; s_1 +\!\!+\; s_2
\end{array}$$

Here $List^{+}\ A$ is the type of non-empty lists containing elements of type A. Finally p $\mathsf{between}$ ns parses strings matching p between the name parts in the non-empty vector ns, returning a vector containing the results from p:

$$\begin{array}{ll}
\mathsf{between}_{[\,]} & : [\,] \in p\ \mathsf{between}\ (t :: [\,]) \;\cdot\; t :: [\,] \\
\mathsf{between}_{::} & : x \in \flat\ p \;\cdot\; s_1 \to xs \in p\ \mathsf{between}\ ts \;\cdot\; s_2 \to \\
 & \quad x :: xs \in p\ \mathsf{between}\ (t :: ts) \;\cdot\; t :: s_1 +\!\!+\; s_2
\end{array}$$

The definition of $\_\mathsf{between}\_$ uses ∞, which marks its argument as being coinductive. It can be read as a suspension, and comes with "force" and "delay" operators:

$$\begin{array}{ll}
\infty & : Set \to Set \\
\flat & : \forall\ \{A\} \to \infty\ A \to A \\
\sharp & : \forall\ \{A\} \to A \to \infty\ A
\end{array}$$

The rest of the *Parser* data type is inductive, so the only way to define cyclic or infinite parsers/grammars is to use $\_\mathsf{between}\_$. Note that because the first and last name parts accepted by p $\mathsf{between}$ ns are the first and last name parts in ns the cycles introduced by $\_\mathsf{between}\_$ are neither left nor right recursive.

Figure 5 contains a precedence graph for the parser combinators. Note that in Agda ordinary function application (juxtaposition), non-operator identifiers, parenthesised expressions and closed operators all bind strictly tighter than every other operator. This amounts to modifying the grammar in Fig. 2 by removing op_{p}^{closed} from $\widehat{p}$ and adding the following productions:

$$expr ::= closed^{+}, \tag{13}$$

$$p\!\uparrow ::= closed^{+}, \quad \text{and} \tag{14}$$

$$closed ::= identifier \mid \underline{(}\ expr\ \underline{)} \mid \bigvee \left\{\, op_{p}^{\text{closed}} \;\middle|\; p \text{ is a graph node} \,\right\}. \tag{15}$$

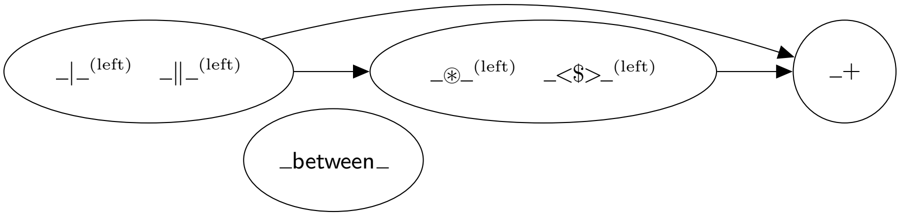

Fig. 5. A precedence graph for the parser combinators.

The non-terminal *identifier* stands for ordinary identifiers; this includes the *names* of operators, like `if_then_else_`, but not operator name parts, unless they are also ordinary identifiers. (Note that this grammar does not correspond exactly to Agda's current expression syntax, partly because it omits constructs like lambda abstractions and dependent function spaces. It should give enough intuition to enable following the examples in the paper, though.)

5.5 The Grammar Scheme

Now the implementation of the grammar scheme can be presented. The definition is yet again given in a module parameterised by a precedence graph:

```
module Mixfix (g : PrecedenceGraph) where
```

A grammar will be defined for g. The grammar is defined by mutual structural recursion/guarded corecursion.

The only delayed parser is *expr*, which corresponds to the non-terminal *expr*:

```
expr : ∞ (Parser (Expr g))
expr = ♯ (precs g)
```

The parser *precs ps* corresponds to $\bigvee \{ \widehat{p} \mid p \in ps \}$, and *inner* (*ops p fix*) corresponds to op_p^{fix}:

```
precs : (ps : List Precedence) → Parser (Expr ps)
precs []        = fail
precs (p :: ps) = (λ (_ , e) → here • e) <$> prec p
                | weakenE                <$> precs ps
inner : ∀ {fix} (ops : List (∃ (Operator fix))) → Parser (In ops)
inner []                 = fail
inner ((_ , op) :: ops)  =
    (λ args → here • args) <$> (expr between nameParts op)
  | weakenI                <$> inner ops
```

Finally we get to *prec p*, which corresponds to the non-terminal $\widehat{p}$. The definition of *prec* follows the grammar scheme given in Sect. 3 closely:

```
prec : (p : Precedence) → Parser (∃ (Ex p))
prec p@(precedence ops sucs) =
    ⟪_⟫     <$>        [closed   ]
  ∥ _⟨_⟩_   <$>  p↑ ⊛ [infx non]    ⊛ p↑
  ∥ appʳ    <$>        preRight + ⊛ p↑
  ∥ appˡ    <$>  p↑ ⊛ postLeft  +
  ∥ fail
  where
  [_] = λ (fix : Fixity) → inner (ops fix)
  p↑  = precs sucs
  preRight : Parser (Out p right → Ex p right)
  preRight = ⟪_⟩_    <$>        [prefx     ]
           | _⟨_⟩ʳ_  <$>  p↑ ⊛ [infx right]
  postLeft : Parser (Out p left → Ex p left)
  postLeft = (λ op    e₁ → e₁ ⟨ op ⟫      )  <$>  [postfx  ]
           | (λ op e₂ e₁ → e₁ ⟨ op ⟩ˡ e₂)  <$>  [infx left] ⊛ p↑
  appʳ = λ fs e → foldr (λ f e → f (similar e)) (λ f → f (tighter e)) fs
  appˡ = λ e fs → foldl (λ e f → f (similar e)) (λ f → f (tighter e)) fs
```

Here $[\mathit{fix}]$ corresponds to op_p^{fix}, $p{\uparrow}$ to $p{\uparrow}$, *preRight* to $\overrightarrow{p}$ and *postLeft* to $\overleftarrow{p}$. Note the use of *foldl* and *foldr* to handle the post-processing of the parse trees. These functions are folds for non-empty lists:

```
foldr : {A B : Set} → (A → B → B) → (A → B) → List⁺ A → B
foldl : {A B : Set} → (B → A → B) → (A → B) → List⁺ A → B
```

The right fold *foldr* applies the argument of type $A \rightarrow B$ to the last element of the list, and the left fold *foldl* applies it to the first element.

The code above is accepted as total by Agda because it uses a lexicographic combination of guarded corecursion and structural recursion: every call path from one definition to itself consists solely of constructors and recursive calls, and either at least one of the constructors is the coinductive constructor ♯, or one argument becomes structurally smaller.

5.6 Correctness

Finally let us show that the grammar scheme is sound and complete with respect to the type of precedence correct expressions, i.e. that the generated expressions are exactly the precedence correct ones.

Due to the precise types used in the definition of the grammar scheme we have already established soundness: all expressions generated by the *expr* nonterminal have to be precedence correct with respect to the relevant precedence graph.

In order to show completeness we first define a function *show* which flattens expressions (the code is omitted here):

$$\mathit{show} : \forall \{ps\} \rightarrow \mathit{Expr}\ ps \rightarrow \mathit{List}\ \mathit{NamePart}$$

We then show, for every expression e, that e is one of the possible results of parsing *show* e:

$$\mathit{complete} : (e : \mathit{Expr}\ g) \rightarrow e \in \flat\ \mathit{expr} \cdot \mathit{show}\ e$$

The proof, which is not included here, is by induction over the structure of e.

6 Implementation

Section 5 describes a (not necessarily efficient) method which, given a precedence graph, parses expressions containing mixfix operators. However, for a programming language with support for user-defined mixfix operators the precedence graph is not predetermined, and different precedence graphs can be in effect at different source locations. If the programming language is defined in a suitable way, then the following procedure can be used to parse a program:

1. Parse the program, treating expressions as flat lists of tokens.
2. Compute the precedence graph in effect for every expression.
3. Parse the flat token lists into real expressions, using the relevant precedence graphs.

This requires that one can identify the extent of an expression without parsing it completely, and also that the relevant precedence graph does not change halfway through an expression. If expressions can bind new operators—consider lambda abstractions, for instance—then the procedure does not quite work, but a reasonable workaround is to include all binding constructs in the “outer” grammar used by the first step above. Note that similar methods are used to parse several existing languages with user-defined infix operators, for instance Haskell.

Using parser combinators to implement the grammar scheme, like in Sect. 5, can be nice: the implementation is almost a direct transliteration of the intended grammar, so it should be easy to understand and modify the code. However, to ensure sufficient efficiency of parsing one has to choose the implementation of the parser combinators (the backend) carefully. There are at least two problems to watch out for:

– The generated grammars are often far from being left factorised.
– The sharing of the precedence DAG might be lost when the DAG is converted into a parser.

If Wadler’s “list of successes” implementation of parser combinators (1985) is used, then one can expect worst-case parse times which are (at least) exponential in the size of the graph, even if the grammar is completely unambiguous. However, as observed by Norvig (1991), inefficient backtracking parsers can be made efficient by using memoisation. We have a prototype implementation which uses memoising parser combinators based on those of Frost and Szydlowski (1996) and memoises the $\widehat{p}$ non-terminals. This means that when the parser backend has found all substrings, starting at a given input position, which match $\widehat{p}$, then the corresponding results and the substrings' endpoints are stored for later reuse. Our experiments indicate that this gives sufficient performance for typical programming language applications, involving limited ambiguity and moderately sized graphs and input strings. If precedence graphs are very large (due to large libraries of operators) it is perhaps a good idea to prune them before parsing, keeping only those parts of the graphs which are relevant based on the name parts present in the current expression.

7 Related Work

Peyton Jones (1986) shows how user-defined mixfix operators can be described using a *fixed* grammar (as opposed to the grammar scheme defined in this work) by distinguishing initial, middle and final tokens of mixfix operators lexically. Support for user-defined precedences is not discussed, and seems hard to incorporate. Peyton Jones argues that operators *should* be lexically distinguishable from other syntactic constructs, to make them easier to parse for humans. We partly agree, but think that this task can be performed by syntax highlighting instead of lexical conventions; in any way, lexical restrictions fit well with the approach developed in this paper. Peyton Jones also discusses how mixfix operator sections can be handled. A section is a partial application of an operator; for instance, the sections `_[ i ]` and `if_then_else y` stand for `λ x → x [ i ]` and `λ b x → if b then x else y`, respectively. Sections can be straightforwardly integrated into our grammar scheme. To avoid a combinatorial explosion of the number of productions one can let the lexer distinguish between hole markers (`_`) placed before, between and after name parts (similarly to Peyton Jones' PRE_TOKEN, IN_TOKEN and POST_TOKEN), and take advantage of this distinction in the grammar scheme.

The work of Aasa (1995, 1992) is probably closest to ours. She shows how a class of possibly ambiguous context-free grammars, whose productions are annotated with disambiguating precedences and associativities, can be translated into unambiguous context-free grammars. She also shows how a parser for user-defined mixfix operators can be implemented using parser combinators. In contrast to our work Aasa only considers total precedence orders. Furthermore her notion of what it means to be precedence correct is more liberal than ours (the strings accepted are claimed to be exactly those which an operator-precedence parser (Floyd 1963; Aho et al. 1986) accepts). This might seem like an advantage, but in order to achieve this result Aasa ends up with an arguably rather complicated grammar scheme, while we have striven to keep the grammars simple and hence easy to understand. The difference lies in how prefix and postfix operators are handled. As an example of how Aasa's system differs from ours, consider the precedence graph in Fig. 6. In our system the string `0 + $ 0` is syntactically incorrect since `$_` binds weaker than `_+_`, whereas Aasa's system accepts arbitrary prefix operators immediately to the right of an infix operator, so in her

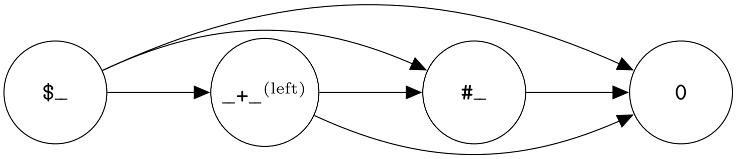

Fig. 6. A precedence graph corresponding to a total precedence order (based on a precedence relation due to Aasa (1995)).

system the string can be unambiguously parsed as 0 + ($ 0). It does not stop there, though. The string # $ 0 + 0, which is also syntactically incorrect in our system, is parsed as # ($ (0 + 0)) in Aasa's. It is accepted because, even though #_ binds tighter than _+_, the occurrence of _+_ is *covered* by $_. Our system has the advantage that one can tell whether a syntax tree is precedence correct by inspecting every node in isolation and considering the relation between the node's operator and the operators of the child nodes. In Aasa's system this is not enough: even though the syntax tree # ($ 0) (where the parentheses indicate the structure of the syntax tree) is precedence correct and #_ binds strictly tighter than _+_ the syntax tree (# ($ 0)) + 0 is not precedence correct.

Several languages and formalisms which support user-defined mixfix operators rely on *type information* to resolve syntactic ambiguities, sometimes in conjunction with user-specified precedences (Pettersson and Fritzson 1992; Missura 1997; Goguen et al. 1999; Paulson et al. 2008). One way to accomplish this is to first parse the input using a possibly ambiguous grammar, then type check every parse tree, and finally accept the input if exactly one parse tree is type correct. Such an approach, while possibly costly, can easily be combined with the grammar scheme described in this work. More optimised approaches are possible, though. The parser for Missura's *higher-order mixfix syntax* (1997), aimed at parsing (linear) mathematical notation, is integrated with Hindley-Milner style type inference.

Missura also argues that precedence relations should not have to be total orders, and Heinlein (2004) argues that precedence relations should be partial orders. The language Fortress uses a non-transitive precedence relation, hard-coded in the language's grammar (Allen et al. 2008).

Finally we note that Missura (1997) and Wansbrough (1999) discuss some other approaches to handling mixfix operators, and that it is also possible to support user-defined *binding constructs*, like ∀x ∈ S. P (Missura 1997; Coq Development Team 2009; Paulson et al. 2008).

8 Conclusions

We have presented a simple grammar scheme for expressions containing mixfix operators, and discussed integration of the grammar scheme into the parsing process of a full-fledged programming language implementation.

The simplicity of our approach comes at a price: other methods, like Aasa's (1995), treat more expressions as being precedence correct, and some methods enable programmers to extend the grammar with other forms of expressions, like binding constructs. However, we believe that our approach strikes a nice balance between simplicity and expressiveness. In the future we plan to replace Agda's support for mixfix operators (Norell 2007) with a grammar scheme and an implementation based on this work.

Acknowledgements

We would like to thank Graham Hutton, Wouter Swierstra and the anonymous reviewers for useful feedback.

References

Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, 1995.

Annika Aasa. *User Defined Syntax*. PhD thesis, Chalmers University of Technology, 1992.

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2009.

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

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., Sam Tobin-Hochstadt, et al. *The Fortress Language Specification, Version 1.0*, 2008.

The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*, 2009.

Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. Draft, 2009.

Robert W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, 1963.

Richard A. Frost and Barbara Szydlowski. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming*, 27(3):263–288, 1996.

Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Introducing OBJ*, 1999.

Christian Heinlein. C+++: User-defined operator symbols in C++. In *INFORMATIK 2004 – Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, volume P-51 of *Lecture Notes in Informatics*, pages 459–468, 2004.

Wafik Boulos Lotfallah. Characterizing unambiguous precedence systems in expressions without superfluous parentheses. *International Journal of Computer Mathematics*, 86(1):1–20, 2009.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18:1–13, 2008.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised Edition*. MIT Press, 1997.

Stephan Albert Missura. *Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing*. PhD thesis, ETH Zürich, 1997.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

Martin Odersky. *The Scala Language Specification, Version 2.7*. Programming Methods Laboratory, EPFL, Switzerland, 2009. Draft.

Lawrence C. Paulson, Tobias Nipkow, and Markus Wenzel. *The Isabelle Reference Manual*, 2008.

Mikael Pettersson and Peter Fritzson. A general and practical approach to concrete syntax objects within ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon L. Peyton Jones. Parsing distfix operators. *Communications of the ACM*, 29 (2):118–122, 1986.

Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd Edition, Advanced Programming Techniques*. MIT Press, 1994.

Philip Wadler. How to replace failure by a list of successes; a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128, 1985.

Keith Wansbrough. Macros and preprocessing in Haskell. Unpublished manuscript, 1999.