Parsing Mixfix Operators

Nils Anders Danielsson (Nottingham)
Joint work with Ulf Norell (Chalmers)

IFL 2008, 2008-09-10

Mixfix (distfix) operators

Infix	_⊢_:_
Prefix	`if_then_else_`
Postfix	_[_]
Closed	⟦_⟧

Useful?

- Can be abused.
- Can enable compact/domain-specific notation.

If used, then ease of parsing for humans is important.

(Agda uses mixfix operators.)

Goal

Mixfix operators should be easy to parse for humans.

Method

- Precedence graph.
- Simple grammar based on graph.

Goal 2

Easy to implement with sufficient efficiency.

Method

- Memoising backtracking parser combinators.

Mixfix operators

Easy to declare (in Agda):

_⊢_:_   _[_]   `if_then_else_`   ⟦_⟧

But what does it mean? How should

$$\emptyset \ , \ \mathtt{n} \ : \ \mathbb{N} \ \vdash \ [\![\ \mathtt{n} \ + \ 11 \ ]\!] \ : \ \mathbb{N}$$

be parsed?

Standard solution: Precedence/associativity.

Precedence and associativity

Precedence	Associativity	Result of parsing `x + y * z`
$+ < *$		`x + (y * z)`
$* < +$		`(x + y) * z`
$+ = *$	Both left	`(x + y) * z`
$+ = *$	Both right	`x + (y * z)`
Otherwise		Parse error

Total order?

No. Why should $\_+\_$ and $\_\wedge\_$ be related?

- Not modular.
- Unnecessary design choices.
- Fewer related operators $\Rightarrow$ parsing easier for humans?

Partial order?

No.

$$\left.\begin{array}{lcl} \_\wedge\_ & < & \_\equiv\_ \\ \_\equiv\_ & < & \_+\_ \end{array}\right\} \Rightarrow \_\wedge\_ < \_+\_$$

Precedence relations

- Directed acyclic graphs.
  - Cyclic graphs often lead to ambiguities.
  - And left (right) recursive grammars.
- One or more operators per node.
- Some operators with associated associativity.
- Note that total and partial orders are DAGs.

Semantics

Given a DAG a context-free grammar is constructed.

Nonterminals:

- *expr* Arbitrary expression.
- $\widehat{i}$ Expression headed by operator from precedence level i.
- $i\!\uparrow$ Expression headed by operator which binds tighter than precedence level i.

$$expr ::= \bigvee \left\{ \widehat{i} \;\middle|\; i \text{ is a graph node} \right\}$$

$$i\!\uparrow ::= \bigvee \left\{ \widehat{j} \;\middle|\; i < j \right\}$$

Semantics

Assume one infix, non-associative, binary operator per node.

$$\widehat{i} ::= i\!\uparrow \; op_i^{\mathrm{non}} \; i\!\uparrow$$

Mixfix

The internal part of an expression:

$$op_i^{\text{non}} \quad ::= \quad op_{i,1}^{\text{non}}\ \mathit{expr}\ op_{i,2}^{\text{non}}\ \mathit{expr}\ \cdots\ op_{i,k}^{\text{non}}$$

Mixfix

Multiple operators with the same precedence:

$$\begin{array}{rcl} op_i^{\text{non}} & ::= & op_{i,1,1}^{\text{non}}\ \mathit{expr}\ op_{i,1,2}^{\text{non}}\ \mathit{expr}\ \cdots\ op_{i,1,k_1}^{\text{non}} \\ & \vdots & \\ & | & op_{i,i_n,1}^{\text{non}}\ \mathit{expr}\ op_{i,i_n,2}^{\text{non}}\ \mathit{expr}\ \cdots\ op_{i,i_n,k_{i_n}}^{\text{non}} \end{array}$$

Postfix

$$\widehat{i} \quad ::= \quad i{\uparrow}\ \ op_i^{\text{postfix}+}$$

Not left recursive, but parse trees need to be post-processed:

$$\mathit{rest}(op \cdots op) \quad \Rightarrow \quad (\cdots(\mathit{rest}\ op)\cdots)op$$

Fold left.

Left associative

$$\begin{array}{rcl} \widehat{i} & ::= & i{\uparrow}\ \ op_i^{\text{postfix}+} \\ & | & i{\uparrow}\ \ (op_i^{\text{left}}\ \ i{\uparrow})^+ \end{array}$$

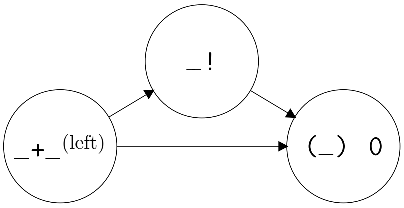
Combined

$$\widehat{i} \quad ::= \quad i{\uparrow}\ \ (op_i^{\text{postfix}} \mid op_i^{\text{left}}\ \ i{\uparrow})^+$$

Full grammar

$$\begin{array}{rcl} \mathit{expr} & ::= & \bigvee \left\{ \widehat{i} \;\middle|\; i \text{ is a graph node} \right\} \\ i{\uparrow} & ::= & \bigvee \left\{ \widehat{j} \;\middle|\; i < j \right\} \\ \widehat{i} & ::= & op_i^{\text{closed}} \\ & | & i{\uparrow}\ op_i^{\text{non}}\ i{\uparrow} \\ & | & (op_i^{\text{prefix}} \mid i{\uparrow}\ \ op_i^{\text{right}})^+\ i{\uparrow} \\ & | & i{\uparrow}\ \ (op_i^{\text{postfix}} \mid op_i^{\text{left}}\ \ i{\uparrow})^+ \\ op_i^{\mathit{fix}} & ::= & \bigvee \{\ p_1\ \mathit{expr}\ p_2\ \mathit{expr}\ \cdots\ p_k \mid \ldots \} \end{array}$$

Example



$$
\begin{aligned}
\textit{expr} &::= \textit{plus} \mid \textit{fac} \mid \textit{closed} \\
\textit{plus} &::= \textit{plus}{\uparrow} \;\; (+ \;\; \textit{plus}{\uparrow})^{+} \\
\textit{plus}{\uparrow} &::= \textit{fac} \mid \textit{closed} \\
\textit{fac} &::= \textit{closed} \;\, !^{+} \\
\textit{closed} &::= (\;\textit{expr}\;) \mid 0
\end{aligned}
$$

Properties

- All name parts unique $\Rightarrow$ unambiguous.
- Neither left nor right recursive.
  - Implemented in the total language Agda.

Implementation

1. Parse the program, treating expressions as flat lists of tokens.
2. Scope checking, fixity declarations.
3. Parse expressions, using the precedence graphs.

Efficiency

Possible performance pitfalls:

- Grammar often far from being left factorised.
- The graph's sharing might be lost.

With *memoising* backtracking parser combinators:

- Simple implementation.
- Sufficient efficiency.

(In prototype.)

Related work

- Lots of work on parsing mixfix operators.
- This particular approach appears new:
  - Directed acyclic graphs.
  - Simple grammar.

Related work

Aasa's work is close to ours, but trades simplicity for more precedence correct expressions.

Assume $\neg\_ < \_\wedge\_$. What about `a ∧ ¬ b`?

- Our approach: No parse since $\_\wedge\_ \not< \neg\_$.
- Aasa: `a ∧ (¬ b)`.

Summary

An approach to mixfix operators which is hopefully easy to understand.

- Precedence graph.
- Simple grammar.
- Simple implementation.

Plan to update Agda's support for mixfix operators.

Questions?

Agda implementation

```
mutual
  data Expr : Set where
    _⟨_⟩_ : forall {assoc} ->
              Expr -> Internal (infx assoc) -> Expr -> Expr
    _⟨_⟫  : Expr -> Internal postfx              -> Expr
    ⟪_⟩_  :         Internal prefx        -> Expr -> Expr
    ⟪_⟫   :         Internal closed              -> Expr

  data Internal (fix : Fixity) : Set where
    _•_ : forall {arity} ->
          Operator fix arity -> Vec Expr arity ->
          Internal fix
```

Agda implementation

```
grammar (node (precedence ops is)) =
    ⟪_⟫              <$>     ⟦ closed   ⟧
  | _⟨_⟩_            <$>  ↑ ⊛ ⟦ infx non ⟧ ⊛ ↑
  | flip (foldr _$_)  <$>      preRight +   ⊛ ↑
  | foldl (flip _$_)  <$>  ↑ ⊛ postLeft +
  where
  ⟦_⟧ = \fix -> internal (ops fix)

  ↑ = ! nodes is

  preRight =  ⟪_⟩_ <$> ⟦ prefx ⟧
           |  _⟨_⟩_ <$> ↑ ⊛ ⟦ infx right ⟧

  postLeft = flip _⟨_⟫                        <$> ⟦ postfx ⟧
           | (\op e₂ e₁ -> e₁ ⟨ op ⟩ e₂) <$> ⟦ infx left ⟧ ⊛ ↑
```