Surface Syntax for Owl-S
** DRAFT 0.9 **

Drew McDermott

September 8, 2004

Change history:

V. 0.91	2004-09-08	Fix some major typos
V. 0.9	2004-09-08:	Provide formal grammar
V. 0.6	2004-07-14:	Shift to infix syntax.

1 Goals

This is a formalization of the surface syntax for processes in Owl-S.

The goals of this exercise are to

1. Provide readable surface syntax
2. Explain how it relates to the usual RDF syntax

The syntax notation is for the Lexiparse parser [McD04]. This document consists of Lexiparse definitions intermixed with text, in a "literate programming" style [Knu84]. See [McD04], appendix 1, for a brief explanation of this incarnation of literate programming.

Here are some examples of process definitions. An atomic process has no body:

```
define atomic process foo(inputs: (x,y - integer),
                          outputs: (xx - String),
                          precondition: loves(x,y),
                          result: (forall (z - integer u,v - string)
                                       purple(x,z) => mauve(y,z)
                                  &
                                  output(xx <= "kool")))
```

The `result` field(s) specify what values are produced and what effects occur when the process is performed by a web-service client.

A composite has a body and no `result`:[1]

```
define composite process baz(outputs: (x - String),
                             inputs: (u,v))
      {perform do_something();
        {
           {
            g :: perform a(n <= v);
            perform foo(x <= u, z <= g.out1)
           }
           ||
           { h :: perform c();
             produce (x <= h.w) }
        }
      }
```

In what follows I will specify the grammar that accepts these definitions (sect. 2), give the parsetrees for them (sect. 3), and then sketch the XMLified RDF that would be produced by "internalizing" the parsetrees.

2 Syntax

The Owl-S grammar is expressed using the Lexiparse formalism. A syntactic token is defined in terms of three components:

1. Its precedence and "fixity" (prefix, suffix, or infix) which includes whether it should be considered as an open bracket (to be matched by a closer). The *parsetree* for an expression is built initially using just this information.

2. Its "tidier," an optional transformation that simplifies a parsetree headed by the token.

3. Its "checkers," which impose constraints on the neighbors (ancestors and descendents) of any sub-parsetree headed by the token after the entire tree has been tidied.

The tidier and checker make use of a simple notation for indicating patterns that might match a parsetree, and, in the case of tidiers, indicate how the tree is to be simplified. This notation is implemented as Lisp macros; in fact all the resources of Lisp are available for tidying and checking (but an elegant grammar will stay within the pattern-oriented subset as much as possible).

[1] [[It's a matter of some controversy whether composite processes should have `result` IOPEs. Since it's Wednesday, I've decided they shouldn't, but the decision can be reversed by deleting the first checker clause for `define-composite`, on p. 3.]]

For example, the following spec for the token `<left-paren>` (produced by lexical analysis from an occurrence of the character `'('`) specifies that it can occur as either a prefix or infix operator, in each case requiring a matched `<right-paren>`. The tidier transforms the prefix version into the token `<group>` and the infix version into the token `<fun-app>`.

```
<<Define left-paren
   ;; Eliminate commas from parenthesized expressions
   (def-syntactic left-paren :prefix (:precedence 0 :context (:open comma right-paren))
                             :infix (:left-precedence 200 :right-precedence 0
                                     :context (:open comma right-paren))
      :tidier
         ((:?  ?(:^+ left-paren [*_] ?@subs)
             !~(:^+ group [*_] ,@subs))
          (:?  ?(:^+ left-paren ?f [_*_] ?@args)
             !~(:^+ fun-app ,f [_*_] ,@args)))) >>
```

(The character group "`:^+`" heads lists that stand for parsetrees. The pattern `?(:^+` *o* —*subs*—`)` *matches* a parsetree with operator *o* and subtrees *subs*, while the construct `!~(:^+` *o* —*subs*—`)` *builds* a parsetree.)

Because this tidier runs right after the parsetree with operator `<left-paren>` is created, so that the token `<left-paren>` is replaced by a more revealing version, no other tidier or checker will ever bother to check for occurrences of it.

2.1 Upper-Level Syntax: Process Definitions

We start by declaring that we're constructing a grammar, called `owl-s`:

```
<<Define grammar-def
(def-grammar owl-s
    :top-tokens (define-atomic define-simple define-composite)
    :lex-parent standard-arith-syntax
    :syn-parent standard-arith-syntax) >>
```

The `:top-tokens` declaration indicates that a legal process definition in Owl-S must be headed by one of the tokens `define-atomic`, `define-simple`, or `define-composite`.

The rest of the grammar consists of definitions of the tokens of the language, starting with those that appear higher (closer to the root) in parsetrees, which are (more or less) the same as the `:top-tokens`.

A process-model definition consists of a sequence of process definitions, each atomic, simple, or composite. All three stem from the `define` operator.

```
<<Define process-definition
   ;; Top node is of form
   ;;     ::= define [atomic | simple] <process-spec>
   ;;         | define composite <process-spec>{ stmt }
   (def-syntactic define :reserved true
```

```
                    :prefix (:precedence 5 :numargs 1)
   :tidier
   ((:?  ?(:^+ define (:^+ composite
                          ?(:^+ process ?name ?@iope-spec)
                          ?@body-spec))
      !~(:^+ define-composite ,name (:^+ iope ,@iope-spec) ,@body-spec))

    (:?  ?(:^+ define (:^+ atomic (:^+ process ?name ?@iope-spec)))
      !~(:^+ define-atomic ,name (:^+ iope ,@iope-spec)))

    (:?  ?(:^+ define (:^+ simple (:^+ process ?name ?@iope-spec)))
      !~(:^+ define-simple ,name (:^+ iope ,@iope-spec)))

    (:else (defect "Unintelligible"))))

(def-checkers define-atomic
    (:up 1 (:?  ?(:~ ?(:^+ ^ (:^+ define-atomic ?@_)))
             (defect "'define atomic' can appear only at top level"))))

(def-checkers define-simple
    (:up 1 (:?  ?(:~ ?(:^+ ^ (:^+ define-simple ?@_)))
             (defect "'define simple' can appear only at top level"))))

(def-checkers define-composite
    (:?  ?(:^+ define-composite ?name (:^+ iope ?@iopes) ?_)
       (cond ((exists (x :in iopes)
                 (matchq ?(:^+ result ?@_)
                         x))
              (defect "'result' IOPE not allowed in composite process "
                      name))
             (t nil)))

    (:up 1 (:?  ?(:~ ?(:^+ ^ (:^+ define-composite ?@subs)))
             (defect "'define composite' can appear only at top level")))) >>
```

Here we have our first checkers, which also use pattern matching. Their output is a list of syntactic *defects*, hopefully empty. The notation `(:up 1 c)` means that the checker c is to run on the *parent* of the current parsetree. The pattern

```
?(:~   ?(:^+      ^      p))
(not) (parsetree) (way high)
```

matches anything that *doesn't* match a parsetree occurring a level above a "real" parsetree matching p. The operator `<^>` is an artifice used only in a "virtual" parsetree above the topmost node of the actual parsetree. So, combined with the `:up` notation, the checker says

"Any parsetree with operator `<define-atomic>` must have no parent except the virtual node above the actual root of the tree; if it does, hang a defect on it."

We analyze `atomic`, `simple`, and `composite` as simple prefix operators, followed by a *process-spec*.

```
<<Define process-classes
   ;; __::= atomic <process-spec>
   (def-syntactic atomic
                  :reserved true
                  :prefix (:precedence 15 :numargs 1))

   ;; __::= simple <process-spec>
   (def-syntactic simple :reserved true
                  :prefix (:precedence 15 :numargs 1))

   ;; __::=  composite <process-spec>... { ...
   (def-syntactic composite :reserved true
                  :prefix (:precedence 15 :numargs 2)
      :tidier
        ((:?  ?(:^+ composite ?proc (:^+ left-brace ?body))
            !~(:^+ composite ,proc ,body))
         (:?  ?(:^+ composite ?proc ?b)
            (defect "Ill-formed body for composite process " proc
                    :% " (must be surrounded by braces):  " b)))) >>
```

The syntax finally gets interesting with the operator `process`:

```
<<Define process-spec

   ;; __::= process <name>(-IOPEs-)
   (def-syntactic process :reserved true
                  :prefix (:precedence 25 :numargs 1)
      :tidier
         ((:?  ?(:^+ process (:^+ fun-app ?name [_*_] (:^+ comma ?@iopes)))
             !~(:^+ process ,name ,@iopes))
          (:?  ?(:^+ process (:^+ fun-app ?name [_*_] ?@subs))
            !~(:^+ process ,name ,@subs))
          (:else (defect "'process' must be followed by <name> and IOPEs in parens")))

      :checkers

         ((:?  ?(:^+ process ?(:~ ?(:+ ?name is-Symbol)) ?@_)
            (defect "Process has nonsymbolic name:  " name))

          (:?  ?(:^+ process ?_ ?@subs)
             (check-all sub subs this-grammar
                  (:?  ?(:^+ ?(:~ ?(:|| inputs outputs locals participants
```

```
                                        precondition result))
                              ?_)
                       (defect "Process has illegal IOPE spec:  " sub))))

        (:?  ?(:^+ process ?_ ?@subs)
           (nconc (occ-range subs 0 1 ?(:^+ local ?@_))
                  (occ-range subs 0 1 ?(:^+ input ?@_))
                  (occ-range subs 0 1 ?(:^+ output ?@_))
                  (occ-range subs 0 1 ?(:^+ precondition ?@_))
                  (occ-range subs 1 :infty ?(:^+ result ?@_))))

        (:up 1 (:?  ?(:~ ?(:^+ ?(:|| atomic simple composite)
                               ?(:^+ process ?@_)))
                  (defect "Process must be declared atomic, simple, or"
                          " composite"))))) >>
```

There are two kinds of IOPEs, the IOs (parameters) and the PEs (preconditions and effects). One key feature of the former is that they are followed by a parameter-declaration list with a distinctive syntax.

<<*Define* param-decls

```
   ;; <IOPE> ::= inputs :  (...)
   (def-syntactic inputs :reserved true
                         :prefix (:context 1 :precedence 120
                                  :local-grammar ((1 typed-var-grammar)))
      :tidier
         (typed-vars-tidier 'inputs)

      :checkers
         ((:up 1 (:?  ?(:^+ iope ?@_)))))

   ;; <IOPE> ::= outputs :  (...)
   (def-syntactic outputs :reserved true
                          :prefix (:context 1 :precedence 120
                                   :local-grammar ((1 typed-var-grammar)))
      :tidier
         (typed-vars-tidier 'outputs)

      :checkers
         ((:up 1 (:?  ?(:^+ iope ?@_)))))

   ;; <IOPE> ::= locals :  (...)
   (def-syntactic locals :reserved true
                         :prefix (:context 1 :precedence 120
                                  :local-grammar ((1 typed-var-grammar)))
```

```
      :tidier
         (typed-vars-tidier 'locals)

      :checkers
      ((:up 1 (:?  ?(:^+ iope ?@_)))))

   ;; <IOPE> ::= participants :  (...)
   (def-syntactic participants :reserved true
                        :prefix (:context 1 :precedence 120
                                  :local-grammar ((1 typed-var-grammar)))
      :tidier
         (typed-vars-tidier 'participants)

      :checkers
         ((:up 1 (:?  ?(:^+ iope ?@_)))) >>
```

```
<<Define typed-vars-tidier
(defun typed-vars-tidier (sort)
   (\\ (this-ptree _)
      (match-cond this-ptree
         (:?  ?(:^+ ?,sort (:^+ colon (:^+ group ?@decls)))
            !~(:^+ ,sort ,@decls))
         (:else (defect "'" sort "' doesn't occur in the form "
                        sort ":(...)")))) >>
```

(The construct

`(\\ (`*—params—*`)` *—body—*`)`

is equivalent to

`#'(lambda (`*—params—*`)` *—body—*`)`

except that ignorable variables can be declared as such by being named "`_`".)

The local grammar `typed-var-grammar` is for typed variable declarations such as `(x,y - String n - Integer)`, in which the hyphens must have precedence lower than that of the commas.

```
<<Define typed-var-grammar
(def-grammar typed-var-grammar
    :lex-parent owl-s
    :syn-parent owl-s
    :replace ((minus hyphen))

:definitions
(
   (def-syntactic \| :infix (:precedence 16 :context :grouping))
```

```
   (def-syntactic left-paren :prefix (:left-precedence 200 :right-precedence 0
                                      :context (:open \| right-paren)))

   ;; hyphens can be generated only by replacement of <minus>
   ;; Precedence puts hyphen above comma in parsetree --
   (def-syntactic hyphen :infix (:precedence 17 :context :binary)
       :tidier
          ;; Eliminate commas --
          ((:?  ?(:^+ hyphen (:^+ comma ?@vars) ?type)
             !~(:^+ hyphen ,@vars ,type)))

       :checkers
          ((:?  ?(:^+ hyphen ?@vars ?_)
             (and (not (is-list-of vars #'is-Symbol))
                  (defect "Vars in declarations are not all symbols:  "
                                  vars)))))
)) >>
```

The token with name "|" is the invisible "contiguity" operator that Lexiparse inserts whenever it expects an operator and none appears in the lexeme stream. In the declaration `(x,y - String n - Integer)`, such an operator will be inserted between `String` and `n`.

The clause `:replace ((minus hyphen))` means to replace the token `<minus>` generated by the lexer with `<hyphen>` when `typed-var-grammar` is in control. That means that even when control returns to the normal `owl-s` grammar, the token derived from "-" will not start looking like a minus again.

The `<colon>` token is yet another prefix operator. It has the odd property that it gets tidied away in the only contexts where it is legal. So the function in its `:checkers` list just checks to see if it actually vanished.

```
<<Define colon
   ;; <name> ::= <name> :  <name>
   ;;
   (def-syntactic colon :infix (:context :binary :precedence 210)
                        ;; -- namespace construct, with high precedence
                        :prefix (:context 1 :precedence 120)
                        ;; -- Occurs only after inputs, outputs, preconditions, effects
                        ;; Low precedence, but higher than comma
      :tidier
         ((:?  ?(:^+ colon ?namespace [_*_] ?name)
             !~(:^+ name-wrt-space ,namespace ,name)))

      :checkers
         ((:up 1 (:?  ?(:^+ ?_ ?@_)
                   (defect "Stray colon")))))) >>
```

The colon is also used as an infix operator. `ecom:interest-rate` means the symbol `interest-rate` in a namespace associated with the prefix `ecom`. Colons of this kind are converted to the token `<name-wrt-space>`, partly so that the checker attached to `<colon>` runs only for the prefix version.

The other two things that can appear in an IOPE list are preconditions and results:

```
<<Define preconds
   ;; <IOPE> ::= precondition :  ...
   (def-syntactic precondition :reserved true
                               :prefix (:context 1 :precedence 120)
      :tidier
         ((:?  ?(:^+ precondition (:^+ colon ?exp))
             !~(:^+ precondition ,exp))
          (:else (defect "'precondition' not followed by ':  <expression>'")))

      :checkers
         ((:up 1 (:?  ?(:^+ iope ?@_))))) >>
```

The formula in the `precondition` field of a process definition has no special syntax, but is just part of the general-purpose logical-expression system that generates all those XML literals at the leaves of Owl-S. This system is discussed in section 2.4.

2.2 The Syntax of `results`

The `result` field(s) are different.

```
<<Define results
   ;; <IOPE> ::= result :  ...
   (def-syntactic result :reserved true
                         :prefix (:context 1 :precedence 120)
      :tidier
      ((:?  ?(:^+ result (:^+ colon ?exp))
          !~(:^+ result ,exp))
       (:else (defect "'result' not followed by ':  <expression>'")))

      :checkers
         ((:up 1 (:?  ?(:^+ iope ?@_)))
          <<Insert:  result-tree-checker (p. ??cf/result-tree-checker??)>>)) >>
```

Results can include several constructs that are not meaningful, or mean something different, in other contexts. Even though a result may look like a predicate-calculus formula, it is not really an entity with a truth value, but a recipe for *changing* the truth values of various propositions. For this reason, I prefer using the verb "impose" to describe a result. A result R does not become "true"; it is *imposed* following an action or event. Here is a rough definition of what that means:

1. To impose an atomic formula p is to make it true
2. To impose a negated atomic formula `not`p is to make p false. For many implementations, this translates into deleting p from a representation of the current situation, so that the use of a closed-world assumption [¡¿] will allow an implementation to conclude that p is false.
3. To impose p`=>`q is to impose q if p was true before the action or event.
4. To impose $\forall$(`x`)p[`x`] is to impose $p[a]$ for all objects a. In practice, most implementations only handle the special case $\forall vars(p[vars]$`=>`$q[vars])$, which means "For every set of values v for *vars* such that $p[v]$ is true before the action or event, impose $q[a]$." Owl-S is typical in singling this case out.
5. To impose `output`(p_1`<=`$v_1, \ldots, p_k$`<=`v_k), where each p_i is an output of the current process, make each v_i the value of p_i. (This notation corresponds to the `withOutput` construct of Owl-S.)

Various syntactic constraints are implied by this definition of "impose." There are others as well: We don't allow disjunctive effects in Owl-S [[do we? what about existential quantifiers?]]. To check that a result is syntactically legal, we must write a straightforward Lisp procedure (called `result-defects`) to walk through a parsetree headed by `<result>`.

First, we define the special operators (`<when>` and `<output>`) that can occur only in results.

```
<<Define lex-when
   ;; '=>' becomes the token <when>
   (def-lexical #\= (dispatch
                      (#\> (token when))
                      (:else (token equals)))) >>

<<Define syn-when
   (def-syntactic when :infix (:context :binary
                               :left-precedence 110 :right-precedence 111)) >>

<<Define lex-bind-param
   ;; '<=' becomes the token <bind-param>
   (def-lexical #\< (dispatch
                      (#\= (token bind-param))
                      (:else (token less)))) >>

<<Define bind-param-syn
   (def-syntactic bind-param :infix (:context :binary
                                     :precedence 110)
      :checkers
         ((:?  ?(:^+ bind-param ?p ?_)
            (and (not (is-Symbol p))
                 (defect "Non-symbol to left of '<=':  " p)))
          <<Insert:  bind-param-context (p. ??cf/bind-param-context??)>>)) >>
```

```
<<Define output-syntax
   (def-syntactic output :reserved true
                  :prefix (:precedence 120 :numargs 1)
      :tidier
         ((:?  ?(:^+ output (:^+ group ?@bindings))
             !~(:^+ output ,@bindings))
          (:else (defect "Output must be followed by bindings in parens")))

      :checkers
         ((:?  ?(:^+ output ?@bindings)
             (check-all b bindings this-grammar
                (:?  ?(:~ ?(:^+ bind-param ?@_))
                   (defect "Outputs must all be of form 'param <= exp', not "
                           b)))))) >>
```

The checker for results calls the recursive procedure `result-defects`:

```
<<Define result-tree-checker
:. (def-syntactic result ...
      ...
      :checkers
         (...  .:
          (lambda (ptree _)
             (match-let ?(:^+ result ?exp)
                        ptree
                (result-defects exp))) >>

<<Define result-syntax-checker
(defun result-defects (exp)
   (let-fun ()
      (match-cond exp
         (:?  ?(:^+ forall ?vars ?r)
            (append (decls-defects vars)
                    (forall-body-must-be-whens r)))
         (:?  ?(:^+ exists ?vars ?r)
            (list (defect "Existential quantifiers are illegal in results")))
         (:?  ?(:^+ when ?_ ?effect)
            (result-defects effect))
         (:?  ?(:^+ and ?@conjuncts)
            (<!  result-defects conjuncts))
         (:?  ?(:^+ group ?elt)
            (result-defects elt))
         (:?  ?(:^+ group ?@_)
            (list (defect "Can't have <comma> as a connective")))
         (:?  ?(:^+ not ?elt)
            (must-be-fun-app elt))
         (:?  ?(:^+ output ?@_)
```

```
            '())
          (t (must-be-fun-app exp)))

    :where

       (:def decls-defects (vl)
           (match-cond vl
              (:?  ?(:^+ group ?@decls)
                 (repeat :for ((decl :in decls))
                    (match-cond decl
                       (:?  ?(:^+ hyphen ?@_)
                            '())
                       (:?  ?(:^+ comma ?@vars)
                          (and (not (is-list-of vars #'is-Symbol))
                               (list (defect "Vars in declarations are not all symbols:  "
                                             vars))))
                       (:?  ?var
                          (and (not (is-Symbol var))
                               (list (defect "Var in declaration is not a symbol:  "
                                             var)))))))
              (:else (defect "Variable declarations must be enclosed in parens"))))

       (:def forall-body-must-be-whens (r)
          (match-cond r
             (:?  ?(:^+ when ?_ ?e)
                (result-defects e))
             (:?  ?(:^+ and ?@conjuncts)
                (<!  forall-body-must-be-whens
                    conjuncts))
             (:else (list (defect "Body of 'forall' must be a '=>' or a conjunction"
                                  " of '=>' expressions")))))) >>

<<Define must-be-atomic
(defun must-be-fun-app (e)
   (match-cond e
      (:?  ?(:^+ fun-app ?_ ?@_)
        '())
      (:else (list (defect "Context requires atomic formula")))))) >>
```

One thing this procedure does not check is whether each occurrence of a variable in a result is bound by a dominating `forall`. That check is deferred until the internalization phase, although it would have been reasonable to do it here. Internalization is managed by a subgrammar called `owl-s-as-rdf`, discussed in section 3.

2.3 The Bodies of Composite Processes

Unlike atomic and simple processes, composite processes have *bodies*, the stuff in left braces after the IOPE specs.

```
<<Define syn-braces
   (def-syntactic left-brace :prefix (:precedence 0
                                      :context (:open nil right-brace)))

   (def-syntactic right-brace :suffix (:left-precedence 0 :context :close)) >>
```

(See the syntactic definition for `composite`, p. 5.) The current grammar handles only a few control constructs, but their definitions should suffice to give the flavor. The constructs in question are `unordered`, `sequence`, `perform`, and `produce`. The first two are denoted by character sequences "`||`" and "`;`". "`perform`" and "`produced` are reserved word. We haven't used `produce` before, but we need a way to indicate how a composite process's outputs get set, and we do it with `produce(`o_1`<=`e_1`, ..., `o_k `<=` e_k`)`, where each o_i is an output of this process.

```
<<Define lex-parallel
:. (def-lexical #\| .:
                    (dispatch
                       (#\| (token parallel)) >>

<<Define lex-semicolon
:. (def-lexical-tokens      .:
                        (#\; semicolon) >>

<<Define control-constructs
   (def-syntactic parallel :infix (:precedence 60 :context :grouping)
      :tidier !'elim-left-braces

      :checkers (control-composition-check
                 (:up 1 control-context-check)))


   (def-syntactic semicolon :infix (:precedence 80 :context :grouping)
      :tidier !'elim-left-braces

      :checkers (control-composition-check
                 (:up 1 control-context-check)))

   (def-syntactic perform :reserved true
                          :prefix (:context 1 :precedence 90)
      :tidier
         ((:?  ?(:^+ perform (:^+ fun-app ?name [_*_] ?@pbs))
             !~(:^+ perform ,name ,@pbs))
          (:else (defect "Unintelligible")))
```

```
   :checkers
      ((:?  ?(:^+ perform ?name ?@_)
          (and (not (is-Symbol name))
               (defect "Perform of process with illegal name " name)))

       (:?  ?(:^+ perform ?name ?@bdgs)
          (check-all b bdgs this-grammar
             (:?  ?(:~ ?(:^+ bind-param ?_ ?_))
                (defect "Illegal data input " b " to perform of " name))))

       (:up 1 control-context-check)))

;; 'produce (p1 <= v1 , ...)'  indicates bindings to outputs of this
;; process
(def-syntactic produce :reserved true
                     :prefix (:context 1 :precedence 90)
   :tidier
      ((:?  ?(:^+ produce (:^+ group ?@bindings))
          !~(:^+ produce ,@bindings))
       (:else (defect "'produce' must be followed by bindings in parens")))

   :checkers
      ((:?  ?(:^+ produce ?@bindings)
          (check-all b bindings this-grammar
             (:?  ?(:~ ?(:^+ bind-param ?@_))
                (defect "'produce' args must all be of form 'param <= exp', not "
                        b))))))  >>
```

Note that the syntax of the parenthesized items after `perform` and `produce` is identical to that of the items after `output`, although they all serve different purposes. In each case, the items must be a sequence of parameter bindings, p`<=`v. (Remember that the characters "`<=`" are converted to the token `<bind-param>`.) For `output` and `produce`, they describe how the results of the current process are set; the former is for atomic and simple processes, and occurs in a `result` construct, while the latter occurs in the body of a composite process.

```
<<Define bind-param-context
:. (def-syntactic bind-param :infix (:context :binary
                                     :precedence 110)
   :checkers
      (...      .:
       (:up 1
          (:?  ?(:^+ ?(:|| output perform produce) ?@_))) :.)).:  >>
```

Any component of a control construct can be tagged. We indicate a tag using a double colon:

```
<<Define lex-colon
   ;; '::'  is for step tags --
   (def-lexical #\:  (dispatch
                       (#\:  (token tag))
                       (:else (token colon)))) >>
```

The syntax is simple:

```
<<Define tag-syn
   (def-syntactic tag :infix (:context :binary :precedence 80)
      :checkers
         ((:?  ?(:^+ tag ?name ?_)
             (and (not (is-Symbol name))
                  (defect "Illegal tag:  " name)))

          (:?  ?(:^+ tag ?_ ?x)
             (match-cond x
                (:?  ?(:^+ ?op ?@_)
                   (cond ((memq op +owl-s-control-operators+)
                          !())
                         (t (defect "Illegal as tagged element:  " x))))))

          (:up 1 control-context-check))) >>
```

The composition and context of the control constructs are governed by the following tidiers and checkers:

```
<<Define control-checks
(defun elim-left-braces (pt _)
   (match-cond pt
      (:?  ?(:^+ ?c ?@subs)
         (multi-let (((okay defects)
                      (repeat :for ((sub :in subs)
                                    :collectors okay defects)
                       :result (values okay defects)
                         (match-cond sub
                            (:?  ?(:^+ left-brace ?@subsubs)
                               (cond ((= (len subsubs) 1)
                                      (one-collect okay
                                         (first subsubs)))
                                     (t
                                      (one-collect defects
                                         (defect "Left-brace encompasses"
                                                 " strange number of subtrees")))))
                            (t (one-collect okay sub))))))

            (cond ((null defects)
```

```
                 !~(:^+ ,c ,@okay))
                (t defects))))
      (t false)))

(defun control-composition-check (ptree g)
   (match-let ?(:^+ ?_ ?@elements)
              ptree
      (check-all e elements g
         (:?  ?(:~ ?(:^+ ?_ ?@_))
            (list (defect "Atomic expression " e " not allowed as"
                          " element of composed process")))

         (:?  ?(:^+ ?c ?@_)
            (cond ((memq c +owl-s-control-operators+)
                   !())
                  (t (list (defect "Illegal as element of composed process:  "
                                   e))))))))

(defun control-context-check (ptree _)
   (match-cond ptree
      (:?  ?(:^+ ?c ?@_)
         (cond ((or (memq c +owl-s-control-operators+)
;;;;                    (eq c 'left-brace)
                    (eq c 'define-composite))
                !())
               (t (list (defect "Control construct in illegal context")))))) >>
```

where

```
<<Define control-ops
(defconstant +owl-s-control-operators+
    '(parallel semicolon tag perform produce)) >>
```

2.4 Formulas

The only parts of Owl-S that are left to deal with are the logical formulas in conditions, effects, and various output constructs. Here we incorporate a simple logical language capturing the normal encodings of Lisp as infix syntax.

Most of this grammar is inherited from the `simple-arith` grammar that is part of the Lexiparse package[[which hasn't been released yet, but there's a description in [McD04]]]. All this grammar does is provide the usual precedence hierarchy for multiplication (*), addition (+), and their ilk. All the precedences for the operators lie in the range 100 to 200.

All we introduce here are the usual quantifiers:

```
<<Define quantifiers
   (def-syntactic forall :reserved true
```

```
                        :prefix (:context 2 :precedence 90
                                 :local-grammar ((1 typed-var-grammar))))

   (def-syntactic exists :reserved true
                        :prefix (:context 2 :precedence 90
                                 :local-grammar ((1 typed-var-grammar)))) >>
```

3 Translation to RDF/XML

The grammar laid out in section 2 yields the parsetrees shown in table 1 for the processes `foo` and `baz` presented in section 1. To convince yourself of this, you can verify that anywhere a parsetree with op_1 appears as an immediate subtree of a parsetree with op_0, either op_1 has higher precedence, or it originally appeared inside brackets of some sort (which may have been tidied away). You can also verify that none of the checkers in section 2 would find any defects.

Having successfully parsed expressions of a language, the next step is to translate the parsed expressions into some internal representation. This process is called *internalization*. There are various possibilities. The most attractive is to produce a plan structure that can be used to implement or reason about a web service. The exercise we focus on here is translation to the XML serialization of RDF. This will relate the surface syntax to the ontology of [Coa04], the latest release of Owl-S.

3.1 Recursive Transformation of Parsetrees

All programs that traverse one recursively defined data structure in order to produce another recursively defined data structure look pretty much alike. So I will only sketch the algorithm. The entire thing can be found at [[]], but if I were you I wouldn't even read the sketch. If you really want to make sense of it, you have to be familiar with the tools documented in [McD03] as well as the parser itself [McD04].

Somewhat surprisingly, there doesn't seem to be a standard set of Lisp data structures for representing XML. The file `[[...]]xml.lisp` is a straightforward rendition of the required abstractions: names relativized to namespaces, and elements (name + attribute list + contents). The internalization algorithm works by producing an XML element corresponding to an Owl-S process. The Lisp pretty-printer is then programmed to print the XML elements out in a readable indented form. In practice, the structure of XML elements is much more likely to be useful than the printed representation (e.g., for producing RDF triples from an XML tree).

Anyway, we start by creating a subgrammar of the basic Owl-S grammar.

```
<<Define def-rdf-grammar
(def-grammar owl-s-as-rdf
    :int-parent owl-s) >>
```

We also define the usual namespaces, omitting most details:

```
<<Define namespace-definitions
```

```
<define-atomic>
   foo
   <iope>
      <inputs>
         <hyphen>
            (x y integer)
      <outputs>
         <hyphen>
            (xx String)
      <precondition>
         <fun-app>
            (loves x y)
      <result>
         <group>
            <forall>
               <group>
                  <hyphen>
                     (z integer)
                  <hyphen>
                     (u v string)
               <when>
                  <fun-app>
                     (purple x z)
                  <and>
                     <fun-app>
                        (mauve y z)
                     <output>
                        <bind-param>
                           (xx "kool")
```

```
<define-composite>
   baz
   <iope>
      <outputs>
         <hyphen>
            (x String)
      <inputs>
         <comma>
            (u v)
   <semicolon>
      <perform>
         (do_something)
      <parallel>
         <semicolon>
            <tag>
               g
               <perform>
                  a
                  <bind-param>
                     (n v)
            <perform>
               foo
               <bind-param>
                  (x u)
               <bind-param>
                  z
                  <dot>
                     (g out1)
         <semicolon>
            <tag>
               h
               <perform>
                  (c)
            <produce>
               <bind-param>
                  x
                  <dot>
                     (h w)
```

Table 1: Parsetrees for Processes `foo` and `baz`

```
(defvar owl-s-namespace*
        (make-XML-namespace
           :global true
           :governor
              (string->uri
                 "http://www.daml.org/services/owl-s/1.1/Process.owl")))

(defvar rdf-namespace*
        (make-XML-namespace ...))

(defvar xsd-namespace*
        (make-XML-namespace ...))
(defvar comlog-namespace*
        (make-XML-namespace ...))

...>>
```

Internalizing Owl-S processes requires internalizing their IOPE declarations and, in the case of composites, their bodies.

```
<<Define process-internalizations
   (def-internal define-atomic (pt _)
      (simple-process-definition pt "AtomicProcess"))

   (def-internal define-simple (pt _)
      (simple-process-definition pt "SimpleProcess"))

   (defun simple-process-definition (pt type-string)
       (match-let ?(:^+ ?_ ?name (:^+ iope ?@iope-specs))
                  pt
          (make-XML-element
             :type (owl-s-name type-string)
             :attributes (list (tuple rdf-ID-name* name))
             :contents (nth-value 2 (iope->xml iope-specs)))))

   (def-internal define-composite (pt _)
      (match-let ?(:^+ define-composite ?name
                       (:^+ iope ?@iope-specs)
                       ?body-exp)
                 pt
         (multi-let (((bound-vars output-vars var-xmls)
                      (iope->xml iope-specs)))
            (make-XML-element
               :type (owl-s-name "CompositeProcess")
               :attributes (list (tuple rdf-ID-name* name))
```

```
            :contents (append var-xmls
                              (list (body->xml
                                       body-exp bound-vars output-vars
                                       (extract-step-tags body-exp))))))))) >>
```

Internalizing the IOPEs requires internalizing declarations, plus conditions and effects.

```
<<Define iope-internalizer

;;; Each element of 'iope-specs' is an 'inputs', 'outputs', etc.
;;; parsetree.
;;; For 'inputs', 'outputs", and 'locals', sus are parsetrees headed
;;;    by <hyphen>, <comma>, or a variable.
;;; Returns < input-n-local-vars, output-vars, xml-elements >
(defun iope->xml (iope-specs)
...)  >>
```

Declaration parsetrees are headed by `<hyphen>` or `<comma>`, or are simple variables. The internalizer for a list of such parsetrees reflects that fact:

```
<<Define vardecl-internalizer
(defun vardecls->xml (labeled-decls var-role)
   (let-fun ()
      (match-let ?(:^+ ?_ ?decls)
                 labeled-decls
         (match-cond decls
            (:?  ?(:^+ hyphen ?@vl ?ty)
               (declarations vl ty))
            (:?  ?(:^+ comma ?@vl)
               (declarations vl false))
            (t
             (declarations (list decls) false))))

    :where

       (:def declarations (vars type)
          (values
             vars
             (repeat :for ((v :in vars))
              :collect
                (make-XML-element ...))))))  >>
```

Preconditions are internalized by turning them into XML literals (see sect. 2.4). Results require analysis in order to resolve them into the standard Owl-S components:

```
<<Define res-tree-analyzer
;;; 'ins' are variables bound as inputs and locals.
;;; 'outs' are variables bound as outputs.
```

```
;;; The latter are the only variables that can appear to the left of
;;; '<='.  Everywhere else a variable must be an element of 'ins' (or
;;; bound by a local quantifier).
(defun res-tree->xml (res ins outs)
   (let-fun ()
      (walk-through res false ins)

    :where

      (:def walk-through (subres must-see-when bvars)
         (match-cond subres
            (:?  ?(:^+ forall ?vars ?r)
               (multi-let (((vl resVars)
                            (decls->resVars vars)))
                  (append resVars
                          (walk-through r true (append vl ins)))))
            (:?  ?(:^+ when ?condition ?effect)
               (cons (logical-expression-element
                         "inCondition" condition bvars !())
                     (effect-elements effect bvars)))
            (:?  ?(:^+ ?(:|| and group) ?@elts)
               (<!  (\\ (e) (walk-through e must-see-when bvars))
                    elts))
            (must-see-when
             (signal-problem res-tree->xml
                "Result contains 'forall' without '=>' to make bindings work:  "
                :% res))
            (t (effect-elements subres bvars))))

       (:def effect-elements (eff bvars) ...))  >>
```

. . . and so forth.

3.2 A Sow's Ear from a Silk Purse: the XML Version

The algorithm I've sketched actually works, at least on these two examples. The resulting XML for process `foo` is shown in table 2; for `baz`, in tables 3 and 4. I've omitted the "front matter" (entity definitions and such) in tables 3 and 4 that duplicates stuff from table 2.

A Owl-S Syntax Skeleton

```
<<Define File owl-s-syn.lisp
;-*- Mode:  Common-lisp; Package:  ytools; Readtable:  ytools; -*-
(in-package :ytools)
```

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
  <!ENTITY owls "http://www.daml.org/services/owl-s/1.1/Process.owl">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY clog "http://www.ihmc.us/users/phayes/SCLJune2004.html">
<rdf:RDF
  xmlns:"owls"="&"owls";#"
  xmlns:"rdf"="&"rdf";#"
  xmlns:"xsd"="&"xsd";#"
  xmlns:"clog"="&"clog";#"
  xmlns="http://www.daml.org/services/owl-s/1.1/Process.owl">

<AtomicProcess
      rdf:ID="foo">
   <hasInput rdf:ID="x"><parameterType rdf:resource="&xsd;#integer"/></hasInput>
   <hasInput rdf:ID="y"><parameterType rdf:resource="&xsd;#integer"/></hasInput>
   <hasOutput rdf:ID="xx"><parameterType rdf:resource="&xsd;#String"/></hasOutput>
   <hasPrecondition
         expressionLanguage="&clog;#CommonLogic"
         rdf:dataType="&xsd;#string">
      (precondition (fun-app loves x y))
   </hasPrecondition>
   <hasResult>
      <Result>
         <resultVar rdf:ID="z"><parameterType rdf:resource="integer"/></resultVar>
         <resultVar rdf:ID="u"><parameterType rdf:resource="string"/></resultVar>
         <resultVar rdf:ID="v"><parameterType rdf:resource="string"/></resultVar>
         <inCondition
               expressionLanguage="&clog;#CommonLogic"
               rdf:dataType="&xsd;#string">
            (fun-app purple x z)
         </inCondition>
         <hasEffect
               expressionLanguage="&clog;#CommonLogic"
               rdf:dataType="&xsd;#string">
            (fun-app mauve y z)
         </hasEffect>
         <withOutput>
            <OutputBinding>
               <toParam rdf:resource="#xx"/>
               <valueFn
                     expressionLanguage="&clog;#CommonLogic"
                     rdf:dataType="&xsd;#string">
                  "kool"
               </valueFn>
            </OutputBinding>
         </withOutput>
      </Result>
   </hasResult>
</AtomicProcess>
```


Table 2: XML for process `foo`

```
<?xml version='1.0' encoding='ISO-8859-1'?>
 ...
<CompositeProcess
      rdf:ID="baz">
   <hasInput rdf:ID="u"/>
   <hasInput rdf:ID="v"/>
   <hasOutput rdf:ID="x"><parameterType rdf:resource="&xsd;#String"/></hasOutput>
   <Sequence
         rdf:parseType="Collection">
      <Perform><process rdf:resource="do_something"/></Perform>
      <Unordered
            rdf:parseType="Collection">
         <Sequence
               rdf:parseType="Collection">
            <Perform
                  rdf:ID="#g">
               <process rdf:resource="a"/>
               <hasDataFrom>
                  <Binding>
                     <theParam rdf:resource="n"/>
                     <valueFn
                           expressionLanguage="&clog;#CommonLogic"
                           rdf:dataType="&xsd;#string">
                        v
                     </valueFn>
                  </Binding>
               </hasDataFrom>
            </Perform>
            <Perform>
               <process rdf:resource="foo"/>
               <hasDataFrom>
                  <Binding>
                     <theParam rdf:resource="x"/>
                     <valueFn
                           expressionLanguage="&clog;#CommonLogic"
                           rdf:dataType="&xsd;#string">
                        u
                     </valueFn>
                  </Binding>
                  <Binding>
                     <theParam rdf:resource="z"/>
                     <valueFn
                           expressionLanguage="&clog;#CommonLogic"
                           rdf:dataType="&xsd;#string">
                        (output-of g out1)
                     </valueFn>
                  </Binding>
               </hasDataFrom>
            </Perform>
         </Sequence>
```

...

Table 3: XML for process baz (part 1)

```
...
         <Sequence
               rdf:parseType="Collection">
            <Perform rdf:ID="#h"><process rdf:resource="c"/></Perform>
            <Produce>
               <output-binding>
                  <Binding>
                     <theParam rdf:resource="x"/>
                     <valueFn
                           expressionLanguage="&clog;#CommonLogic"
                           rdf:dataType="&xsd;#string">
                        (output-of h w)
                     </valueFn>
                  </Binding>
               </output-binding>
            </Produce>
         </Sequence>
      </Unordered>
   </Sequence>
</CompositeProcess>
```

Table 4: XML for process baz (part 2)

```
(depends-on %parser/ parser arith %hacks/ xml)

<<Insert:  grammar-def (p. ??cf/grammar-def??)>>

(defvar owl-s-lexical-chars* '(#\_))

<<Insert:  control-ops (p. ??cf/control-ops??)>>

<<Insert:  typed-vars-tidier (p. ??cf/typed-vars-tidier??)>>

<<Insert:  result-syntax-checker (p. ??cf/result-syntax-checker??)>>

<<Insert:  must-be-atomic (p. ??cf/must-be-atomic??)>>

<<Insert:  control-checks (p. ??cf/control-checks??)>>

(with-grammar owl-s

;;; LEXICAL

   (def-lexical-tokens ((#\{ left-brace)
```

```
                        (#\} right-brace)
                        <<Insert:  lex-semicolon (p. ??cf/lex-semicolon??)>>
                        (#\.  dot)))

   (def-lexical #\|
                :. (dispatch .:
                    <<Insert:  lex-parallel (p. ??cf/lex-parallel??)>>
                       (:else (token or))))

   ;; '->' is ordinary if-then --
   (def-lexical #\- (dispatch
                       (#\> (token imply))
                       (:else (token minus))))

   <<Insert:  lex-when (p. ??cf/lex-when??)>>

   <<Insert:  lex-bind-param (p. ??cf/lex-bind-param??)>>

   <<Insert:  lex-colon (p. ??cf/lex-colon??)>>

   (def-lexical (#\_ #\a - #\z #\A - #\Z #\?)  (lex-sym owl-s-lexical-chars*)
       :name alphabetic)


;;; SYNTACTIC

   <<Insert:  process-definition (p. ??cf/process-definition??)>>

   <<Insert:  process-classes (p. ??cf/process-classes??)>>

   <<Insert:  process-spec (p. ??cf/process-spec??)>>

   <<Insert:  param-decls (p. ??cf/param-decls??)>>

   <<Insert:  preconds (p. ??cf/preconds??)>>

   <<Insert:  results (p. ??cf/results??)>>

   <<Insert:  output-syntax (p. ??cf/output-syntax??)>>

   <<Insert:  colon (p. ??cf/colon??)>>

   <<Insert:  quantifiers (p. ??cf/quantifiers??)>>

   <<Insert:  syn-when (p. ??cf/syn-when??)>>
```

```
   <<Insert:  syn-braces (p. ??cf/syn-braces??)>>

   <<Insert:  bind-param-syn (p. ??cf/bind-param-syn??)>>

   <<Insert:  control-constructs (p. ??cf/control-constructs??)>>

   <<Insert:  tag-syn (p. ??cf/tag-syn??)>>

   <<Insert:  left-paren (p. ??cf/left-paren??)>>

   (def-syntactic dot :infix (:precedence 205 :context :binary)
      :checkers
      ((:?  ?(:^+ dot ?step ?output)
          (nconc (cond ((is-Symbol step) !())
                       (t (list (defect "Illegal name to left of dot in "
                                        step "." output))))
                 (cond ((is-Symbol output) !())
                       (t (list (defect "Illegal name to right of dot in "
                                        step "." output))))))))
)


;;; LOCAL GRAMMAR for type declarations

<<Insert:  typed-var-grammar (p. ??cf/typed-var-grammar??)>>>>
```

B Owl-S Internalization Code for RDF/XML

```
<<Define File owl-s-rdf.lisp
;-*- Mode:  Common-lisp; Package:  ytools; Readtable:  ytools; -*-
(in-package :ytools)

;;; INTERNALIZATION A: Translation to RDF

<<Insert:  def-rdf-grammar (p. ??cf/def-rdf-grammar??)>>

(defun string->uri (s)
   (net.uri:intern-uri
      (net.uri:parse-uri s)))

<<Insert:  namespace-definitions (p. ??cf/namespace-definitions??)>>

(defvar rdf-ID-name*
```

```
        (make-XML-name
           :string "ID" :namespace rdf-namespace*))

(defvar rdf-resource-name*
        (make-XML-name :string "resource" :namespace rdf-namespace*))

(with-grammar owl-s-as-rdf

   <<Insert:  process-internalizations (p. ??cf/process-internalizations??)>>

)

<<Insert:  iope-internalizer (p. ??cf/iope-internalizer??)>>

<<Insert:  vardecl-internalizer (p. ??cf/vardecl-internalizer??)>>

<<Insert:  res-tree-analyzer (p. ??cf/res-tree-analyzer??)>>

;;; Returns < vars, resVar-XML-list >
(defun decls->resVars (vars)
...)

...

(defun body->xml (body bvars output-vars step-tags) ...)

...>>
```

References

[Coa04] The Owl-S Coalition. *Owl-S Release 1.1beta*. Available at `http://www.daml.org/services/daml-s/1.1beta`, 2004.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal* , 27(2):97–111, 1984.

[McD03] Drew McDermott. *The YTools Manual*. Available at`http://www.cs.yale.edu/~dvm/papers/ytdoc.pdf`, 2003.

[McD04] Drew McDermott. *Lexiparse: A Lexicon-based Parser for Lisp Applications*. Draft, available at `http://www.cs.yale.edu/homes/dvm/papers/parser-manual.pdf`, 2004.