

OberonDoc: A Document Generator for the Oberon Language
- A Preliminary Report -



www.kaykonrad.de/einsommernachtstraum.html

Victor Winter (with special thanks to Ralf Lämmel and Felix
Friedrich)

Outline

Motivation and Problem Identification

Example

Demo

What is Currently Supported

A Specification of the HTML Generation Algorithm

Overview

Links and the Generation of Unique Names

The Parsing Spectrum

Outline

Motivation and Problem Identification

Example

Demo

What is Currently Supported

A Specification of the HTML Generation Algorithm

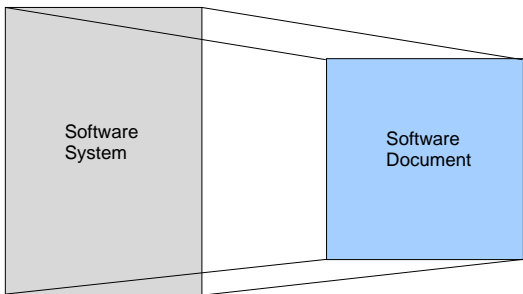
Overview

Links and the Generation of Unique Names

The Parsing Spectrum

Documentation

The primary goal of documentation is to provide a *description* of a (software) system *suitable* for a given *audience*.



Remark: We restrict ourselves to the construction of documents whose information content does not exceed the original system.

Effective Documentation

In order to be *effective*, documentation should facilitate reasoning about a (targeted) set of questions. In other words, the document should be constructed in such a way that it helps the audience obtain answers to questions it finds interesting.

An Abstract Model of a Document

The *paper form* of a document can be modeled as a sequence of information elements:

$$\mathcal{I} = \langle i_1, i_2, \dots, i_n \rangle$$

The purpose of a particular sequence is to help the reader develop an understanding that lets them answer a set of questions:

$$\mathcal{Q} = \{q_1, \dots, q_k\}$$

Views of Information

Assumption: Given an initial sequence \mathcal{I} and a question $q_r \in \mathcal{Q}$, the reader must “rapidly” *form* and *traverse* (e.g., in their mind) the sequence $\mathcal{I}_{q_r} = \langle i_{r_1}, \dots, i_{r_m} \rangle$ in order to answer q_r . Let us use the term *document view* when referring to sequences like \mathcal{I}_{q_r} .

When \mathcal{I} is stored electronically, there are a number of mechanisms that can be used to generate or otherwise support document views – either in part or in full.

- ▶ HTML *links*
- ▶ Expandable sections of text
- ▶ Dropdown menus
- ▶ Popups

Information Sequence Metrics

Metrics and properties can be postulated for the purposes of predicting the effectiveness of an information sequence \mathcal{I}_q in answering a given question $q \in \mathcal{Q}$.

- ▶ The size of \mathcal{I}_q should be minimal (e.g., irrelevant information should be abstracted away).
- ▶ It may be useful to show (in some fashion) how \mathcal{I}_q relates to \mathcal{I} (e.g., how the elements of \mathcal{I}_q are embedded in \mathcal{I}).

Moving On...

A primary driver of document generation is the set \mathcal{Q} . To determine \mathcal{Q} , it helps to know the target audience.

So what is the target audience for OberonDoc?

The Target Audience for OberonDoc

1. An Oberon programmer *using* services provided by a collection of modules (e.g., types and procedures exported from a library).
2. An Oberon programmer *modifying* (e.g., maintaining or evolving) the source code of a collection of modules.
3. Other...

Information of Interest

User of Services:

- ▶ A *listing* of the modules in the collection.
- ▶ A *description* of the *exported elements* of each module in the collection (e.g., interface information). Specifically, the descriptions of
 - ▶ *exported constants* – consists of the constant declaration plus natural language comments
 - ▶ *exported types* – consists of the type declaration plus natural language comments
 - ▶ *exported variables* – consists of the var declaration plus natural language comments
 - ▶ *exported procedures* – consists of the signature plus natural language comments

Information of Interest

Maintainer of Software:

- ▶ A *module listing*.
- ▶ A description of *all* the elements of each module in the collection.
- ▶ Additional candidates for inclusion:
 - ▶ *scope tree* – A tree structure showing the distinct scopes within a module.
 - ▶ *visibility* – A listing of what is visible within a given scope.
 - ▶ Other refactoring-related information.

Outline

Motivation and Problem Identification

Example

Demo

What is Currently Supported

A Specification of the HTML Generation Algorithm

Overview

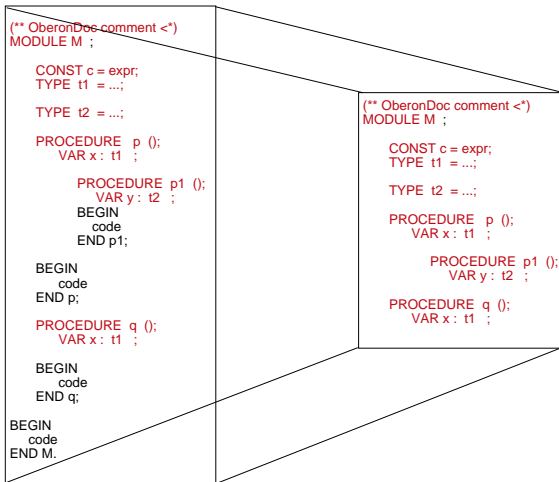
Links and the Generation of Unique Names

The Parsing Spectrum

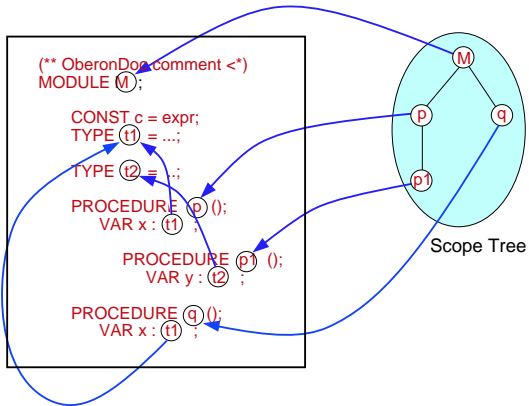
A Source Code View of a Simple Oberon Module

```
(** OberonDoc comment <*)  
MODULE M ;  
  
  CONST c = expr;  
  TYPE t1 = ...;  
  
  TYPE t2 = ...;  
  
  PROCEDURE p ();  
    VAR x: t1 ;  
  
      PROCEDURE p1 ();  
        VAR y: t2 ;  
        BEGIN  
          code  
        END p1;  
  
    BEGIN  
      code  
    END p;  
  
  PROCEDURE q ();  
    VAR x: t1 ;  
  
    BEGIN  
      code  
    END q;  
  
  BEGIN  
    code  
  END M.
```

Reducing Information



Links that Support Document Views



Outline

Motivation and Problem Identification

Example

Demo

What is Currently Supported

A Specification of the HTML Generation Algorithm

Overview

Links and the Generation of Unique Names

The Parsing Spectrum

Views Currently Supported

The present implementation supports the following views:

View: *Scope-Tree*. For each module, a *scope tree* is displayed (in outline form) showing the nesting relationships between all environments in the module.

View: *Visible-Declarations*. For each *environment*, a *view* is constructed showing all *declarations* within the module that are *visible*. That is, elements explicitly declared within this environment as well as elements declared in outer environments are shown. IMPORTANT: *shadowing* of declarations is also shown.

Navigation

Views are, in part, supported by a basic hypertext construct called a *link*. Specifically, links are mechanism that can be used to establish connections between Web resources.

` anchor ` → ` anchor `

- ▶ *Inter-View Navigation*: From the *scope-tree view* it is possible to *navigate* to the *visible-declarations view* associated with a particular environment.
- ▶ *Intra-View Navigation*: From a *visible-declarations view* it is possible to navigate from any type *use* to the corresponding type *def*.

Demo



Outline

Motivation and Problem Identification

Example

Demo

What is Currently Supported

A Specification of the HTML Generation Algorithm

Overview

Links and the Generation of Unique Names

The Parsing Spectrum

The goal here is to provide an incomplete *sketch* an OberonDoc algorithm having the following properties:

1. The syntactic analysis infrastructure needed is *minimal* (e.g., perhaps only some form of enhanced lexical analysis instead of full-blown parsing).
2. The algorithm is suitable for *implementation* in an imperative framework (e.g., Oberon).
3. The algorithm may lend itself to a *concurrent* implementation.

Process 0

Precondition: Develop a parsing/lexical analysis function capable of translating the source code of an Oberon module into an internal form.

$$M.mod \rightarrow S_M$$

The internal form S_M should be *traversable* and it should be possible to recognize (and manipulate) the following:

- ▶ Low-Level: recognition of various *tokens* and *token types*
- ▶ High-Level: Oberon *declarations* and *special comments*

Remark

Recognition of *declarations* is simplified if one assumes that Oberon modules are *normalized* (e.g., no nested record declarations, etc.).

Aside: Traversals

Assumption

It is assumed that a top-down left-to-right traversal of S_M corresponds to the lexical order of the source $M.mod$.

Construction of Stacks

Traverse S_M in a top-down left-to-right fashion and construct a *DeclarationStack* and a *FrameStack*.

- ▶ The contents of the *DeclarationStack* can be used to construct the set of all declarations visible from within a given scope.
- ▶ The contents of the *FrameStack* can be used to construct unique HTML-link names the set of all type identifiers visible from within a given scope.

Remark

The DeclarationStack and FrameStack can be merged. However, the discussion then becomes a bit more involved.

Aside: More on Stacks

Conceptually, a stack can be modeled as a *list* of *elements*. In such a model, a *push* operation places a new element on the front of the list, and a *pop* will remove the element from the front of the list.

In the context of this specification, stack *elements* are used to store information (explicitly) contained within the declaration section of a *named scope*. The *stack structure* itself then models the *nesting* of named scopes.

Process 1

For each *distinct scope* within a module, construct a *list of declarations (LOD)* visible from within the scope. This list should include:

- ▶ All elements *explicitly declared* within the scope.
- ▶ All elements *implicitly declared*. That is, all elements declared in outer scopes. This should include *shadowed* elements (i.e., elements that have been re-declared in the given scope).

Process 1 - Operational Details

LODs can be inductively constructed by *traversing* nested scopes in a *top-down (left-to-right)* fashion.

- ▶ *First*, complete the construction of the DeclarationStack *element* for the given (i.e., current) scope.
- ▶ *Then* construct the *LOD* for the current scope by composing (e.g., concatenating) all the declarations in the DeclarationStack.

Output:

$$S'_M \xrightarrow{\text{Process 1}} \langle LOD_1, LOD_2, \dots, LOD_n \rangle$$

Process 2

For each *distinct scope* within a module, construct a *list of types (LOT)* visible from within the scope. This list should include:

- ▶ All types *explicitly declared* within the scope.
- ▶ All types *implicitly declared*. That is, all types declared in outer scopes. This should include *shadowed* type declarations (i.e., types that have been re-declared in the given scope).

Process 2 - Operational Details

*LOT*s can be inductively constructed by *traversing* nested scopes in a *top-down (left-to-right)* fashion.

- ▶ *First*, complete the construction of the FrameStack *element* for the given (i.e., current) scope.
- ▶ The *LOT* for the current scope then consists of all the declarations in the FrameStack.

Output:

$$S'_M \xrightarrow{\text{Process 2}} \langle LOT_1, LOT_2, \dots, LOT_n \rangle$$

Intermediate Result

Given:

$$S_{LOD} = \langle LOD_1, LOD_2, \dots, LOD_n \rangle$$

$$S_{LOT} = \langle LOT_1, LOT_2, \dots, LOT_n \rangle$$

Form the following set of tuples:

$$\mathcal{D} = \{ \langle LOD_i, LOT_i \rangle \mid LOD_i = \#i(S_{LOD}) \wedge LOT_i = \#i(S_{LOT}) \}$$

We now describe how the set \mathcal{D} can be processed.

Process 3

Given $(LOD_i, LOT_i) \in \mathcal{D}$. Using information in the LOT_i , *rewrite* type identifiers occurring in LOD_i to *linked-identifiers*. Let us refer to the resulting (i.e., the rewritten) structure as LOD'_i .

$$LOD_i \xrightarrow{LOT_i} LOD'_i$$

- ▶ Rewriting must conform to a *globally adhered-to naming scheme* and must be consistent across all modules.
- ▶ Suitable naming can be accomplished via a *FrameStack* (which will be discussed shortly).

Process 4

- ▶ Use the elements in an *LOD'* to populate a set of HTML tables.
- ▶ *Compose* HTML tables into well-formed HTML documents (e.g., add boilerplate front-ends, etc.) . The resulting documents should support the scope-tree and visible-declarations views described earlier in this presentation.
- ▶ For each *LOD'* generate a distinct HTML file.

Links

- ▶ A key component of HTML document creation is the *consistent* and *appropriate* generation of *links*.
- ▶ In this specification, suitable *link generation* is realized in terms of a collection of operations on a *frame stack*.

Aside: Input Assumptions and Requirements

- ▶ This specification assumes that a file containing an Oberon module can be decomposed into a sequence of tokens.
- ▶ It is also assumed that a suitable parsing/lexical analysis infrastructure is in place to enable the *recognition* of certain *token types*.

Aside: Basic Input Assumptions

We assume the following:

- ▶ A *file* (e.g., a Oberon module) is modeled as a sequence of *tokens*.
- ▶ A token sequence may be traversed multiple times (though this requirement can be removed).
- ▶ The token sequence is *traversed* from *left-to-right* (i.e., in lexical order). One reason for this is to maintain appropriate information on the frame stack.

Aside: Token Type Recognition

- ▶ *isNamedScope* : *token* \rightarrow *BOOLEAN* this function returns true if the token denotes a named scope (e.g., `MODULE M ...`).
- ▶ *isDef* : *token* \rightarrow *BOOLEAN* this function returns true if the token denotes an identifier occurring in a declaration context (e.g., `TYPE T = ...`).
- ▶ *isUse* : *token* \rightarrow *BOOLEAN* this function returns true if the token denotes a type identifier occurring in a use context (e.g., `VAR ... : T`).
- ▶ *isScopeExit* : *token* \rightarrow *BOOLEAN* a boolean function that when given a token returns true if the token is the keyword “*END*” (denoting the end of a named scope).

Basic Definitions: Environments and Coordinates

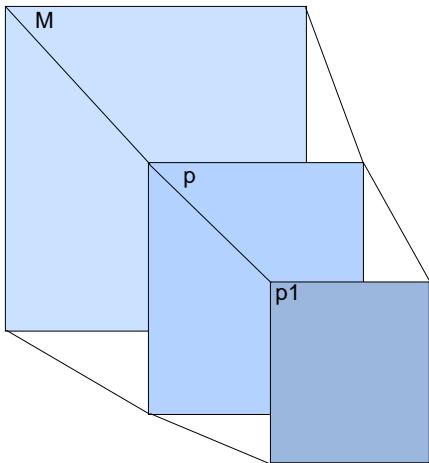
- ▶ A *module* or *procedure* constitutes a *named scope*.
- ▶ An *environment* (\mathcal{E}) can be uniquely designated (i.e., given a unique name) by a “dot-separated” sequence of identifiers where each identifier in the sequence corresponds to a named scope.

$$id_1.id_2.\dots.id_n$$

- ▶ Let us call such a designation the *coordinate* of \mathcal{E} .

Example

coordinate = M.p.p1



```
(** OberonDoc comment <*)  
MODULE M ;  
  
  CONST c = expr;  
  TYPE t1 = ...;  
  
  TYPE t2 = ...;  
  
  PROCEDURE p ();  
    VAR x : t1 ;  
  
    PROCEDURE p1 ();  
      VAR y : t2 ;  
  
  PROCEDURE q ();  
    VAR x : t1 ;
```

Basic Definitions: Frames

- ▶ A *frame* is a tuple consisting of a coordinate together with the list of identifiers (explicitly) declared within the scope.

$$F_i = (\text{coordinate}_{\varepsilon_i}, \text{def_list}_i)$$

- ▶ A *frameStack* is a list of frames where the head of the list (e.g., F_i) denotes the *current frame*.

$$\begin{aligned} \text{frameStack} &= [F_i, F_{i-1}, \dots, F_0] \\ \text{currentFrame} &= F_i \end{aligned}$$

Some Useful Frame and Stack Operations

`addDef(x, (coord, xs) :: stack) = (coord, (x::xs) :: stack)`

`popStack(frame :: stack) = stack`

`pushFrame(coord, stack) = (coord, []) :: stack`

`makeCoord(id, stack) = getCoord(stack) . id`

`getCoord([]) = OberonDoc`

`getCoord((coord,_) :: stack) = coord`

Stack Lookup

```
getDefCoord(token,(coord, xs) :: stack) =  
  if isMember(token, xs) then coord  
  else getDefCoord(token, stack)
```

Maintenance of the Frame Stack

In the context of a top-down left-to-right traversal, the frame stack can be maintained as follows:

```
token → pushFrame(coordinate, stack)
if { isNamedScope(token) andalso
    coordinate = makeCoord(token, stack) }
```

```
token → popStack(stack)
if { isScopeExit(token) }
```

```
token → addDef(token, stack)
if { isDef(token) }
```

Construction of Def Links

In the context of a top-down left-to-right traversal, identifiers can be rewritten to linked-identifiers as follows:

token



```
<a name="coordinate.html#coordinate.token"> token </a>  
if { isDef(token) andalso  
    coordinate = getCoord(stack) }
```

Construction of Use Links

In the context of a top-down left-to-right traversal, identifiers can be rewritten to linked-identifiers as follows:

```
token → <a href="use.html#def.token"> token </a>
if { isUse(token) andalso
    use = getCoord(stack) andalso
    def = getDefCoord(token,stack) }
```

OR

```
token → <a href="def.html#def.token"> token </a>
if { isUse(token) andalso
    def = getDefCoord(token,stack) }
```

Outline

Motivation and Problem Identification

Example

Demo

What is Currently Supported

A Specification of the HTML Generation Algorithm

Overview

Links and the Generation of Unique Names

The Parsing Spectrum

The Parsing Spectrum

- ▶ Lexical Analysis
- ▶ Fuzzy Parsing
- ▶ Island Grammars
- ▶ Skeleton Grammars
- ▶ Error Repair
- ▶ Precise Parsing

For Further Reading I



Steven Klusener and Ralf Lämmel.

Deriving tolerant grammars from a base-line grammar.

Proceedings of the International Conference on Software Maintenance (ICSM'03).