JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Abstract Syntax Trees in XML

BAKKALAUREATSARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaureus/Bakkalaurea der technischen Wissenschaften

in der Studienrichtung

INFORMATIK

Eingereicht von:
David Tanzer, 0055020

Angefertigt am:
Institut für Systemsoftware

Betreuung:
o. Univ.-Prof. Dr. Hanspeter Mössenböck
Dipl.-Ing. Markus Löberbauer

St, Peter in der Au, September 2006

Table of Contents

Abstract

In the last few years, XML has become the method of choice for all kinds of structured data. There are many benefits of using XML to define a data representation. This can be compared to the introduction of standard containers in transport and logistics: They made easy interaction of all involved parties possible. To have a common format for data representation like XML increases the interoperability of computer programs in the same way.

There are many standard ways of dealing with XML, for example one can find open source XML parsers for almost all programming languages. In many modern programming languages XML support is even built-in. With XSL Transformations there is a standard way of transforming XML to other formats, and Document Type Definitions of XML Schema Definitions allow it to define the structure of an XML document.

In this thesis XML is used to represent the "Abstract Syntax Tree" of a compiler. The abstract syntax tree is an internal representation which is generated by the parser of a compiler when it analyzes a source file. This abstract syntax tree is highly structured so it can be easily represented by an XML document which has a tree structure too.

The first chapter of this thesis contains a short introduction and a description of the tools used to create the programs. The next chapter describes a compiler for the component pascal language which outputs its internal syntax tree almost directly to an XML file ("intermediate file format"). While the intermediate file format is very specific to the component pascal programming language, the third chapter defines a more general XML file format for abstract syntax trees. The fourth chapter shows how the intermediate file format is transformed to the general form using XSL Transformations.

1 Introduction

1.1 Programming in XML

Although XML is designed to represent structured data, other people have thought about representing program code in XML over the years too. There are already some fully XML based programming languages like SuperX++ or Water (which will be discussed later in this chapter) and there are mixed approaches as well (like Java Server Pages, where code can be embedded in HTML pages).

The idea behind creating an XML based programming language is that XML is fully customizable ([XML04]) which means that you can create your own set of tags which is appropriate to the data you want to store. Another advantage is that data and program code can be stored in the same XML repository.

There are many standard ways of processing XML files. XML Schema ([XSD04]) is used to define the structure of a specific file format, with XSL Transformations ([XSLT99]) XML files can be transformed to other formats and there are standard data models for XML parsers like the Document Object Model (DOM).

More information about the advantages can be found in [WILSON03]. Sean McGrath wrote about some disadvantages of XML based programming systems in [MCGRATH03]. The main disadvantage is that XML makes the readability of programs worse while adding no real advantages. According to him, if you use XML (which is a data description language) to create a programming language "the result is every bit as unsatisfactory as taking a programming language and using it as a data description language".

1.1.1 SuperX++

SuperX++ is an XML based object oriented programming language invented 2001 by Kimanzi Mati. The advantages of SuperX++ are:

- The code is written in XML and can therefore be stored in the same XML repository as the data it operates on.
- Source code written in SuperX++ is subject to the same manipulation techniques that can be performed on ordinary XML data.
- XML messaging and Web Services can easily be created since SuperX++ already uses XML

Syntax.

More information about SuperX++ can be found in [EL04].

1.1.2 Water

Water ([CM04]) is a programming language for rapidly prototyping XML Web services. It is not based on XML but on ConciseXML ([CM03]). ConciseXML is a proprietary markup language which looks somewhat like XML (according to the vendor it is a superset of XML). Since the ConciseXML synax is not as strict as XML constructs like

```
2.<plus 3/>
```

or even

```
<vector "zero" "one" "two"/>.<get 1.<plus 1/>/>
```

can be written in water. This also means that standard XML tools like a DOM parser or an XSLT processor will not work on Water programs.

1.2 Used Tools

The compilers which were created for this thesis rely on the following tools:

- The compiler generator “Coco/R”
- The dependency management system “gc-rr”
- The XML library frontend “gc-xml”
- A XSTL processor (like for example Xalan-J).

1.2.1 Coco/R

Coco/R is a compiler generator which creates a scanner and a parser for a language which is defined as Attributed Grammar [MÖSSENBÖCK05]. For this thesis, the Java – version of Coco/R was used, which means the scanner and the parser generated by Coco/R are Java classes.

The parser and scanner created by Coco/R can be used to create a compiler by simply adding a main class which calls the parser and semantic classes which provide the functionality called within the grammar rules.

The “cp-xastc” - Compiler (which is described in the chapter “Compiling Component Pascal to XML with Coco/R” on page 9) directly outputs its syntax tree to an XML file, so it basically only

needs the gc-xml library (described on page 7) and a symbol table as semantic classes.

1.2.2 gc-rr and gc-xml

The "Guglhupf Commons Resource Repository" (gc-rr) is a tool for managing dependencies of resources required for a Java – project [PIRRINGER06]. It provides an easy way of specifying which resources are needed for a certain project and manages these resources. This means it automatically downloads the right version of all resources to a local repository if this has not already been done, adds them automatically to the classpath of ant – builds, creates the correct classpath for eclipse projects and it can run Java – applications with the correct classpath set.

The dependencies of a resource are specified as an XML file, a *resource specifier*. This file contains informations about the resource itself (because the resource itself might be a dependency of another resource), about the available versions of the resource and the dependencies of each version.

Dependencies between resources are resolved correctly. For example, the cp-xastc compiler (which is described in the chapter "Compiling Component Pascal to XML with Coco/R" on page 9) has a dependency to "gc-xml" which in turn depends on "Dom4J", but the dependency to "Dom4J" does not have to be specified in the resource descriptor for cp-xastc, it will be resolved automatically by gc-rr.

"Guglhupf Commons XML" (gc-xml) is a convenience library for XML processing. It is only a front end for other XML processing libraries like Dom4J. The default implementation uses Dom4J as back end. The advantage of using gc-xml instead of using Dom4J directly is that it provides a small, light-weight interface for XML processing and is very easy to use. Also, the back end could be switched if this is desired.

1.3 Installing and running the compilers

1. Download gc-rr from http://dev.guglhupf.net/commons/rr/index.html and install it.
2. Download the sources of the compilers created for this thesis from http://dev.guglhupf.net/xast/index.html

1.3.1 Compiling and running cp-xastc

1. Compile the sources of the cp-xast compiler using Apache Ant (http://ant.apache.org).
2. To run the compiler use gc-rr:

```
gc-rr run -in [input file].cp -out [output file].xml
```

You can call “gc-rr run” without specifying a resource specifier file because there is only one resource specifier file in the cp-xastc directory (XASTC.rs.xml) and gc-rr will use the information from this file to run the compiler.

1.3.2 Compiling and running xastc

To run the XSLT – stylesheet which transforms the output of cp-xastc to XAST you just call the XSLT processor. Anyway, there is a simple Java program which indents the created XAST file that can be run using gc-rr. To compile and run xastc do the following:

1. Compile the sources of IndentXML using Apache Ant (simply call “ant” in the xastc directory).
2. Download an XSLT processor (like Apache Xalan: http://xalan.apache.org/)
3. Call the stylesheet transformation and IndentXML from the xastc root directory:

```
java -cp [xalan-classpath] -jar [xalan-jarfile] -XSL src/cpast2xast.xsl
     -IN [input].xml -OUT [output].ast
gc-rr run
```

2 Compiling Component Pascal to XML with Coco/R

Component Pascal is a refinement of the Oberon-2 programming language, created by H. Mössenböck and N. Wirth ([MÖSSENBÖCK93]). Component Pascal was created by Oberon microsystems, Inc. The Component Pascal programming language is specified in the Component Pascal Language Report ([OBERON01]).

The Component Pascal compiler which was created as part of this thesis outputs its syntax tree almost directly as an XML file. This is not yet the generic abstract syntax tree file format (XAST) which will be created from the compiler output with an XSL transformation (See “Transformation of the compiler output to XAST”, page 31). The reason for separating the translation into two steps was to have a better insight into how the translation is done.

A description of the compiler generator Coco/R can be found in the “Tools” - chapter on page 6.

2.1 Component Pascal grammar

The complete grammar of Component Pascal as described in [OBERON01] can be found in Appendix B (page 47). Several parts of this grammar had to be re-written to avoid LL(1) – warnings and errors. Some of the alternatives in the grammar can not be evaluated without knowing the context. This section describes the changes to the grammar which were necessary so it could be used as input for Coco/R.

2.1.1 Import List

The import list was originally:

```
39:  ImportList = "IMPORT" [ident ":="] ident {"," [ident ":="] ident} ";".
```

This had to be changed to:

```
43:  ImportList = "IMPORT" ident [ ":=" ident] {"," ident [ ":=" ident]}
     ";".
```

Because of “LL1 warning in ImportList: ident is start & successor of deletable structure”.

2.1.2 Declaration Sequence

In the Declaration Sequence the keyword “PROCEDURE” was the start of 2 alternatives (ProcDecl and ForwardDecl):

```
40:     DeclSeq = { "CONST" {ConstDecl ";" } | "TYPE" {TypeDecl ";"}
41:               | "VAR" {VarDecl ";"}}
42:               {ProcDecl ";" | ForwardDecl ";"}.
```

So it was changed to:

```
44:     DeclSeq = { "CONST" {ConstDecl ";" } | "TYPE" {TypeDecl ";"}
45:               | "VAR" {VarDecl ";"}}
46:               {Procedures ";"}.
47:     Procedures = "PROCEDURE" (ProcDecl | ForwardDecl).
```

Where the two alternatives have been combined and the keyword “PROCEDURE” only appears in the combination of them.

2.1.3 Procedure Declarations

In the original grammar as described in the component pascal language report, the method attributes “NEW”, “ABSTRACT”, “EMPTY” and “EXTENSIBLE” are defined in the following way:

```
46:     ProcDecl = "PROCEDURE" [Receiver] IdentDef [FormalPars]
47:                ["," "NEW"] ["," ("ABSTRACT" | "EMPTY" | "EXTENSIBLE")]
48:                [";" DeclSeq ["BEGIN" StatementSeq] "END" ident].
```

In this grammar, the “,” is the start of both alternatives, so a LL(1) parser can not decide which alternative should be taken. There is another problem with this grammar: It doesn’t express the fact that “ABSTRACT” and “EMPTY” procedures do not have a body and all others have one. When the parser encounters the “;” symbol and a following declaration sequence it can not decide if the declaration sequence belongs to the procedure body or to the surrounding code.

In the following example, the parser will encounter the Symbols “ABSTRACT” and “;”. After that it will find a valid declaration sequence (the declaration of “bar”) and so it will get into the option “[";" DeclSeq ["BEGIN" StatementSeq] "END" ident]” of “ProcDecl” and encounter an error later:

```
1:PROCEDURE (this : SomeClass) foo() : INTEGER, ABSTRACT;
2:PROCEDURE (this : SomeClass) bar() : INTEGER;
3:     (* ... *)
4:END bar;
```

To address these problems the “ProcDecl” rule has been split up into several rules. The method attributes are now treated separately and the part for the procedure body now only follows procedures which are neither “ABSTRACT” nor “EMPTY”:

```
55:     ProcDecl = [Receiver] IdentDef [FormalPars] ("," MethodAttributes
56:                | ProcedureBody).
57:     MethodAttributes = "NEW" NewAttribute | InheritanceAttributes.
58:     NewAttribute = ProcedureBody | "," InheritanceAttributes.
59:     InheritanceAttributes = "ABSTRACT" | "EMPTY" | "EXTENSIBLE"
                                ProcedureBody.
60:     ProcedureBody = ";" DeclSeq ["BEGIN" StatementSeq] "END" ident.
```

2.1.4 Forward Declarations

Forward declarations are translated to the intermediate XML file format by this compiler, but they will be completely ignored by the XSLT style sheet later. This is because in the XAST file format the declaration order does not matter so we do not need any forward references.

2.1.5 Statement

In the “Statement” - rule were two alternatives which started with “Designator”. These have been combined to a new rule.

```
 61:     Statement = [ Designator ":=" Expr
 62:                 | Designator ["(" [ExprList] ")"]
```

This has been changed to:

```
 75:     Statement = [
 76:                 Designator DesignatedStmt
...
 90:     DesignatedStmt = (":=" Expr)
 91:                    | ([ActualParams]).
...
113:     ActualParams = "(" [ExprList] ")".
```

The "(" [ExprList] ")" part has been moved to a separate rule since it is needed later in the “Designator” - rule again.

2.1.6 Designator

The “Designator” - rule contains several problems for this compiler. The original grammar was:

```
 89:     Designator = Qualident {"." ident | "[" ExprList "]" | " ^ " | "$"
 90:                  | "(" Qualident ")" | "(" [ExprList] ")"}.
```

First of all we have the problem that “(“ is the start of two alternatives: "(" Qualident ")" (a type guard) and "(" [ExprList] ")" (a procedure call). Even if we combine these two alternatives we have the same problem again, because then “ident” is the start of both alternatives. The Component Pascal Language Report says about type guards “A *type guard v(T)* asserts that the dynamic type of *v* is *T* (or an extension of *T*), i.e. program execution is aborted, if the dynamic type of *v* is not *T* (or an extension of *T*).” [OBERON01]. This means that the sequence "(" ident ")" is a type guard if the identifier is a type and otherwise it’s a procedure call, so the compiler would have to know the current context to decide which alternative to use. I decided to leave away the type guard – part from this rule so the parser will always parse "(" Qualident ")" as procedure call. This is not so much of a problem since the purpose of this work is not to provide a fully featured component pascal compiler but to show that it’s possible to represent abstract syntax trees in XML.

The next problem we have here is that Component Pascal allows it to completely leave away the actual parameters (including "(" and ")") when you call a procedure that doesn't take any parameters. For example you can write

```
1:Console.WriteLn;
```

and it's treated as a procedure call. This again means that the compiler has to know the context to treat this correctly (it has to know if "WriteLn" is a procedure or a variable within "Console"). Again this has been left away (i.e. the parser will always treat it as a variable or field access) for the same reason as described above.

Note that according to this grammar it is also allowed to write multiple actual parameter lists after a qualified identifier, like:

```
1:foo("a")("b", "c")("d");
```

but I think that does not really make sense. The parser created for this project will parse this and simply add multiple "callProcedure" elements to the designator. Note that the "Gardens Point Component Pascal" - compiler (http://www.plas.fit.qut.edu.au/gpcp/) reports this as an error.

The resulting grammar for "Designator" is:

```
112:    Designator = Qualident {"." ident | "[" ExprList "]" | "^" | "$" |
                    ActualParams}.
113:    ActualParams = "(" [ExprList] ")".
```

2.1.7 Qualident

A qualified identifier in component pascal is an optional module name followed by an identifier:

```
 93:    Qualident = [ident "."] ident.
```

When the optional part is the first part of this rule we again have the problem that "ident" is the start of both alternatives, so this had to be changed. Also the keyword "NEW" is not only used as a symbol in component pascal, it is sometimes used as an identifier too because it is a predeclared procedure which allocates memory for a pointer type.

Another problem is that the compiler can not decide if the first identifier is a module name or not without knowing the context, but this is not really a problem here because both alternatives will be translated into a "select" element (see page 18).

The resulting grammar for qualified identifiers is:

```
117:    Qualident = (ident | "NEW") ["." (ident | "NEW")].
```

2.2 The Compiler

The Compiler is generated with the compiler generator Coco/R which is described in more detail in the "Used Tools" - section of this document (page 6). The XML file is generated using gc-xml which is a wrapper library for XML processing tools like dom4j (page 7).

The compiler has only two source files, XASTCompiler.java contains the main program which initializes and invokes the compiler and ComponentPascal.atg is the compiler description from which Coco/R generates the compiler. The files Scanner.frame and Parser.frame are needed by Coco/R too but they have been simply copied from the Coco/R distribution.

2.2.1 XASTCompiler.java

This file contains the main class for the compiler. It is responsible for parsing the command line arguments, it creates the XML document which will be written by the compiler and it initializes and runs the compiler.

2.2.2 ComponentPascal.atg

This file is the input file for the compiler generator Coco/R. It contains all the productions for the Component Pascal programming language and embedded Java code, which will be executed when the compiler comes to this point in an input file. The files Scanner.java, Parser.java and ErrorStream.java are generated by Coco/R from this source file.

Most of the productions get an XML element as input and add to it another element which corresponds to the production type. The result of this is that the internal syntax tree of the compiler is (almost) directly mapped to the XML file. The code for the production "ProcDecl" (procedure declaration) is for example:

```
112: Procedures<Element parent> (. indent++; debug("Procedures"); .)
113:   = "PROCEDURE"           (. Element procElem =
                                  parent.addElement("procedure"); .)
114:   ( ProcDecl<procElem> | ForwardDecl<procElem>)
115:   (. indent--; .).
...
153: ProcDecl<Element parent> (. indent++; debug("ProcDecl"); .)
154:   =
155:   [Receiver<parent>]      (. IdentDefT procName=new IdentDefT(); .)
156:   IdentDef<procName>      (. parent.addAttribute("name", procName.name);
157:                              if(procName.export)
158:                                parent.addAttribute("export", "true");
159:                              if(procName.readOnly)
160:                                parent.addAttribute("implementOnly",
                                                       "true");
```

```
161:                            .)
162:    [FormalPars<parent>]
163:    ("," MethodAttributes<parent> | ProcedureBody<parent>)
164:    (. indent--; .).
```

The Java – Code in the first line of each production (“**indent++;** debug("Procedures")**;**” plus “**indent++;** debug("ProcDecl")**;**”) and before the end of the productions (“**indent--;**”) is for debugging purposes only. “**IdentDefT**” is a Java class that stores information about identifier definitions.

The production for “Procedures” creates a new XML element as a child of the element “parent”: **Element procElem = parent.**addElement("procedure")**;**. This XML element is then passed to the production for “ProcDecl” which adds some attributes to this element. It then passes the same XML element to the productions “MethodAttributes” and “ProcedureBody” which will again add elements and attributes to it.

3 Abstract Syntax Trees in XML: XAST

The XAST file format has been developed to be a more general representation of an abstract syntax tree than the file format which is directly generated by the compiler described in chapter 2 (page 9). Elements which are not needed in the XML representation (like the ones in expressions which result from operator bind level rules) are simply left away (for details see the section “Expressions” of this chapter on page 27).

The goal for this XML file format was to keep it as general as possible so that it could be used for any programming language. For example there is no assumption about how scopes are nested, this is explicitly expressed in the file format (See “Scopes” on page 17). Anyway there are still some constructs left which correspond to concepts in component pascal (like the “with” element, see page 26), but this should be no limitation for the XAST file format.

The file format itself is an XML file format [XML04] specified as XML Schema Definition [XSD04]. The complete XML Schema for the XAST file format can be found in Appendix A (page 37).

3.1 Attributes

The following XML attributes are used in some of the XML elements later:

```
520:     <xs:attribute name="name" type="xs:string"/>
521:     <xs:attribute name="scope" type="xs:string"/>
522:     <xs:attribute name="prefix" type="xs:string"/>
523:     <xs:attribute name="access">
524:         <xs:simpleType>
525:             <xs:restriction base="xs:string">
526:                 <xs:enumeration value="public"/>
527:                 <xs:enumeration value="package"/>
528:                 <xs:enumeration value="protected"/>
529:                 <xs:enumeration value="private"/>
530:             </xs:restriction>
531:         </xs:simpleType>
532:     </xs:attribute>
533:     <!-- read-only is not the same as constant - it comes from component
534:          pascal, where it's possible to export a variable read-only (so
535:          it's a normal variable in the module/class/... where it is
536:          defined, and read-only everywhere else.
537:     -->
538:     <xs:attribute name="read-only" type="xs:boolean"/>
539:     <xs:attribute name="type" type="xs:string"/>
540:     <xs:attribute name="parameter-type">
541:         <xs:simpleType>
542:             <xs:restriction base="xs:string">
```

```
543:                 <xs:enumeration value="input"/>
544:                 <xs:enumeration value="output"/>
545:                 <xs:enumeration value="input-output"/>
546:             </xs:restriction>
547:         </xs:simpleType>
548:     </xs:attribute>
549:     <xs:attribute name="new" type="xs:boolean"/>
550:     <xs:attribute name="empty" type="xs:boolean"/>
551:     <xs:attribute name="abstract" type="xs:boolean"/>
552:     <xs:attribute name="extensible" type="xs:boolean"/>
553:     <xs:attribute name="select" type="xs:string"/>
```

3.2 General Structure

The root element of a XAST file is “xast-module”. It has a required attribute “name” and contains exactly one “scope” element (For a detailed description of scopes see the section “Scopes” on page 17). Before the “scope” an arbitrary number of “import-module” elements is allowed. The “import-module” element has a required “name” - attribute and indicates that functionality from the module with the given name is needed in this module.

Example:

```
   4:<xast-module xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:noNamespaceSchemaLocation="[Path-To-Schema]/xast.xsd"
                  name="TestAll">
   5:   <import-module name="CPmain"/>
   6:   <import-module name="Console"/>
   7:   <import-module name="ProgArgs"/>
   8:   <scope name="TestAll.global">
...
1124:   </scope>
1125:</xast-module>
```

The additional attributes in “xast-module” (line 4) are only for validation of the XML document against the XML Schema “xast.xsd”.

3.2.1 Module Initializer and Finalizer

Modules can have an Initializer (“module-initializer” element) and a Finalizer (“module-finalizer” element). The Initializer contains code which is executed when the module is loaded, the Finalizer contains code which is executed when the module is unloaded. Both elements contain exactly one “scope” element which contains a “body” element.

3.3 Scopes

In XAST there are no assumptions made about which statements have their own scope or how nested scopes inherit / overwrite symbols from their outer scopes. All these cases are explicitly defined using the “scope” and “using-scope” elements. The “select” element is used to get symbols from the current scope.

3.3.1 “scope” - Element

The “scope” element can contain several different elements because it can occur in many different places in the XAST file. Depending on where the “scope” element occurs not all of the allowed nested elements make sense. The definition of the “scope” element is:

```
15:<xs:element name="scope">
16:  <xs:complexType>
17:    <xs:sequence>
18:      <xs:element ref="using-scope" minOccurs="0"
                     maxOccurs="unbounded"/>
19:      <xs:element ref="constant" minOccurs="0" maxOccurs="unbounded"/>
20:      <xs:element ref="type" minOccurs="0" maxOccurs="unbounded"/>
21:      <xs:element ref="variable" minOccurs="0" maxOccurs="unbounded"/>
22:
23:      <xs:element ref="function" minOccurs="0" maxOccurs="unbounded"/>
24:      <xs:element ref="method" minOccurs="0" maxOccurs="unbounded"/>
25:
26:      <xs:element ref="parameters" minOccurs="0" maxOccurs="1"/>
27:      <xs:element ref="return-type" minOccurs="0" maxOccurs="1"/>
28:      <xs:element ref="this-pointer" minOccurs="0" maxOccurs="1"/>
29:      <xs:element ref="body" minOccurs="0" maxOccurs="1"/>
30:
31:      <xs:element ref="module-initializer" minOccurs="0" maxOccurs="1"/>
32:      <xs:element ref="module-finalizer" minOccurs="0" maxOccurs="1"/>
33:    </xs:sequence>
34:
35:    <xs:attribute ref="name" use="required"/>
36:  </xs:complexType>
37:</xs:element>
```

For example, the “module-initializer” and the “module-finalizer” elements can only occur within the “scope” element of an “xast-module”.

The “scope” element has a required “name” - attribute that assigns a unique name to the scope. This name has to be unique throughout all modules. The XSLT transformation which is described in chapter 4 (page 31) prepends all scope names with the module name and “.global” to guarantee this. The scope for a method of a class contains the class name and the method name too, for example:

```
69:<scope name="TestAll.global.SomeClass.foo">
```

3.3.2 “using-scope” - Element

Every scope can contain “using-scope” elements which indicate that all the Symbols from another scope are reachable within the current scope too. This has been introduced because not all programming languages use the same rules for nesting scopes (in PHP for example the symbols from the global scope are reachable in local scopes only when they have been imported using the “global” keyword).

The “using-scope” element has a required “name” - attribute which contains the name of the scope from which the symbols are used. The element can have an optional “prefix” - attribute which contains a prefix for all symbols in this scope. For a description of how symbols can be accessed refer to the documentation of the “select” element.

3.3.3 “select” - Element

“select” gets a symbol from the current scope or selects a prefix for nested “select” elements. The “select” element has a required “name” - attribute which contains the name of the symbol or prefix to select. Consider the following example code which contains only the “scope”- and “using-scope” elements:

```
 1:<?xml version="1.0" encoding="ISO-8859-1"?>
 2:
 3:<xast-module name="select-example">
 4:    <import-module name="Console"/>
 5:    <scope name="select-example.global">
 6:        <using-scope scope="Console.global" prefix="Console"/>
 7:        <!-- ... -->
 8:        <scope name="select-example.global.Foo">
 9:            <using-scope scope="select-example.global" prefix="global"/>
10:            <!-- ... -->
11:            <scope name="select-example.global.Foo.doSomething">
12:                <using-scope scope="select-example.global.Foo"/>
13:                <using-scope scope="Console.global" prefix="Console"/>
14:                <!-- ... -->
15:            </scope>
16:        </scope>
17:    </scope>
18:</xast-module>
```

From the most inner scope the symbol “WriteLn” from the scope “Console.global” can be selected with the following “select” elements:

```
20:<select name="Console">
21:    <select name="WriteLn"/>
22:</select>
23:
24:<select name="global">
25:    <select name="Console">
```

```
26:         <select name="WriteLn"/>
27:     </select>
28:</select>
```

The first select statement works because the scope “select-example.global.Foo.doSomething” directly uses the scope “Console.global” with the prefix “Console”. The second select works because the scope “select-example.global.Foo.doSomething” uses the scope “select-example.global.Foo” without prefix, and this scope uses the scope “select-example.global” with the prefix “global” and so on.

A symbol which comes from a scope and is directly or indirectly used and where never a prefix is defined can be selected directly. For example, if a symbol “doSomethingElse” is defined in the scope “select-example.global.Foo” than it can be accessed with the following “select” element:

```
30:<select name="doSomethingElse"/>
```

3.4 Types, Functions and Methods

3.4.1 “type” - Element

A type element has the optional attributes “name”, “read-only” and “access”. It contains exactly one of the following Elements: “simple-type”, “array-type”, “pointer-type”, “class” and “function-type”.

“simple-type” elements only contain a “select” element:

```
 1:<type>
 2:     <simple-type>
 3:         <select name="xs:integer"/>
 4:     </simple-type>
 5:</type>
```

“pointer-type” elements have the same attributes and content as “type” elements, since a pointer type represents a pointer to another type:

```
22:<type name="ASimplePointerType" access="private" read-only="false">
23:     <pointer-type>
24:         <simple-type>
25:             <select name="xs:integer"/>
26:         </simple-type>
27:     </pointer-type>
28:</type>
```

A “function-type” element contains exactly one “parameters” element and one “return-type” element:

```
42:<type name="AProcedureType" access="private" read-only="false">
43:     <function-type>
44:         <parameters/>
45:         <return-type>
```

```
 46:             <simple-type>
 47:                 <select name="void"/>
 48:             </simple-type>
 49:         </return-type>
 50:     </function-type>
 51:</type>
```

An “array-type” element contains a “length” element which contains an “expression”. It also contains an “element-type” element which has the same attributes and content like the “type” element.

```
  7:<type>
  8:     <array-type>
  9:         <length>
 10:             <expression>
 11:                 <literal type="xs:integer">5</literal>
 12:             </expression>
 13:         </length>
 14:         <element-type>
 15:             <simple-type>
 16:                 <select name="char"/>
 17:             </simple-type>
 18:         </element-type>
 19:     </array-type>
 20:</type>
```

A “class” element may contain “baseclass” elements (multiple inheritance is allowed in XAST) and must contain a “scope” element. The “baseclass” elements contain exactly 1 “select” element:

```
 30:<type name="DerivedClass" access="public" read-only="true">
 31:     <class>
 32:         <baseclass>
 33:             <select name="SomeClass"/>
 34:         </baseclass>
 35:         <scope name="TestAll.global.DerivedClass">
 36:             <using-scope scope="TestAll.global"/>
 37:             <!-- ... -->
 38:         </scope>
 39:     </class>
 40:</type>
```

3.4.2 “function” and “method” - Elements

Both elements contain exactly one “scope” element. This “scope” element contains the elements “return-type”, “parameters”, “body” and “this-pointer”. Empty or abstract methods do not have a “body” element. The “this-pointer” element is necessary because in Component Pascal it is required to assign a name to the “this” - pointer.

The “return-type” element has the same attributes and content as “type” elements. The “this-pointer” element is empty but has a “name” - attribute.

Both elements have the following required attributes: “name”, “access” and “read-only”. The “method” element can optionally have the following attributes too: “new”, “empty”, “abstract” and “extensible”.

3.4.3 “parameters” and “parameter” - Elements

The “parameters” element can occur in the “scope” of functions and methods. It contains none or more “parameter” elements. The “parameter” element contains either a “type” element (if it is a formal parameter) or an “expression” element (if it is an actual parameter). Formal parameters have two attributes: “parameter-type” which indicates whether it is an input-, output- or input/output-parameter and a “name”.

3.4.4 “variable” and “constant” - Elements

The “variable” element contains exactly one “type” element, the “constant” element contains exactly one “value” element. Both elements have the required attributes “name”, “access” and “read-only”. The “value” element contains exactly one “expression” element.

3.4.5 “body” - Element

The “body” element is a “all-statements”-type, so it can contain all Statement – elements as described in the section “Statements” on page 22.

3.4.6 An example method

```
 98:<method name="foo" access="public" read-only="false">
 99:  <scope name="TestAll.global.DerivedClass.foo">
100:    <using-scope scope="TestAll.global.DerivedClass"/>
101:    <parameters/>
102:    <return-type>
103:      <simple-type>
104:        <select name="xs:integer"/>
105:      </simple-type>
106:    </return-type>
107:    <this-pointer name="this"/>
108:    <body>
...
136:    </body>
137:  </scope>
138:</method>
```

3.5 Statements

3.5.1 Definition of the type “all-statements”

```
210:<xs:complexType name="all-statements">
211:  <xs:sequence>
212:    <!-- declarations... -->
213:    <xs:element ref="constant" minOccurs="0" maxOccurs="unbounded"/>
214:    <xs:element ref="type" minOccurs="0" maxOccurs="unbounded"/>
215:    <xs:element ref="variable" minOccurs="0" maxOccurs="unbounded"/>
216:    <xs:element ref="function" minOccurs="0" maxOccurs="unbounded"/>
217:
218:    <!-- statements -->
219:    <xs:choice minOccurs="0" maxOccurs="unbounded">
220:      <xs:element ref="assignment"/>
221:      <xs:element ref="call"/>
222:      <xs:element ref="if"/>
223:      <xs:element ref="switch"/>
224:      <xs:element ref="while"/>
225:      <xs:element ref="do-while"/>
226:      <xs:element ref="for"/>
227:      <xs:element ref="with"/>
228:      <xs:element name="exit">
229:        <xs:complexType/>
230:      </xs:element>
231:      <xs:element name="return">
232:        <xs:complexType>
233:          <xs:choice minOccurs="0" maxOccurs="1">
234:            <xs:element ref="expression"/>
235:          </xs:choice>
236:        </xs:complexType>
237:      </xs:element>
238:    </xs:choice>
239:  </xs:sequence>
240:</xs:complexType>
```

The “exit” element is used to exit the execution immediately. It is empty and has no attributes. The “return” element has no attributes, but it can contain an “expression” element which is the return value if it occurs within a function. If it is used in a procedure (i. e. a function with return type “void”) it is an empty element with no attributes.

3.5.2 “assignment” - Element

The “assignment” element contains exactly two other elements: “target” and “value”. The “target” element contains a “select” - Element (see page 18) and the “value” element contains an “expression” element as described in the section “Expressions” on page 27.

3.5.3 “call” - Element

The “call” element is used to call functions and methods. It contains an element called “method” and this element contains a “select” element to select the method from the current scope. After that it contains a “parameters” element as described on page 21. The nested “parameter” elements are used as actual parameters which means they contain “expression” elements which calculate the parameter values. The call “someProcedure(x, 17);” would be translated to XAST as:

```
464:<call>
465:  <method>
466:    <select name="someProcedure"/>
467:  </method>
468:  <parameters>
469:    <parameter>
470:      <expression>
471:        <select name="x"/>
472:      </expression>
473:    </parameter>
474:    <parameter>
475:      <expression>
476:        <literal type="xs:integer">17</literal>
477:      </expression>
478:    </parameter>
479:  </parameters>
480:</call>
```

3.5.4 “if” - Element

The “if” - Element represents if - statements. It contains a “condition” element, a “then” element, zero or more “else-if” elements and an optional “else” element. The “condition” element contains an expression as described in the section “Expressions” on page 27. The “then” and the “else” element contain a sequence of statements as described in this section (starting on page 22).

The “else-if” element contains a “condition” element and a “then” element as described above. The following XAST code example shows the usage of if statements:

```
481:<if>
482:  <condition>
483:    <expression>
          <!-- some boolean expression -->
492:    </expression>
493:  </condition>
494:  <then>
        <!-- some statement sequence -->
517:  </then>
518:  <else-if>
519:    <condition>
520:      <expression>
            <!-- some boolean expression -->
```

```
529:       </expression>
530:     </condition>
531:     <then>
           <!-- some statement sequence -->
554:     </then>
555:   </else-if>
556:   <else>
         <!-- some statement sequence -->
579:   </else>
580:</if>
```

3.5.5 “switch” - Element

The “switch” element contains a “condition” element, zero or more “case” elements and an optional “default” element, where the “condition” element contains an expression and the “default” element contains a sequence of statements as described in this section (starting on page 22).

The “case” elements contain zero or more “label” elements and a “body” element. The “body” element contains a sequence of statements which is executed when the case is selected, the “label” elements specify ranges for the value in the “condition” element for which this case will be selected.

A “label” element contains either a “value” element (which contains an expression) or it contains a “range” element which contains two nested elements: “from” and “to” (both contain an expression). The following example shows how “switch” elements are used in XAST:

```
616:<switch>
617:   <condition>
618:     <expression>
           <!-- some expression which yields an integer value -->
620:     </expression>
621:   </condition>
         <!-- ... more cases ... -->
         <!-- the next case has 2 labels, one of them specifies a range -->
663:   <case>
664:     <label>
665:       <value>
666:         <expression>
               <!-- some expression which yields an integer value -->
668:         </expression>
669:       </value>
670:     </label>
671:     <label>
672:       <range>
673:         <from>
674:           <expression>
                 <!-- some expression which yields an integer value -->
676:           </expression>
677:         </from>
678:         <to>
```

```
679:           <expression>
                 <!-- some expression which yields an integer value -->
681:           </expression>
682:         </to>
683:       </range>
684:     </label>
685:     <body>
         <!-- a sequence of statements -->
692:     </body>
693:   </case>
694:   <default>
         <!-- a sequence of statements -->
701:   </default>
702:</switch>
```

3.5.6 “while” - Element

This element represents a while – loop (a conditional loop where the condition is checked before each run). It contains a “condition” element followed by an optional “body” element which are used in the same way as in “if” - statements. The “expression” in the “condition” element has to yield a boolean value, the statement sequence in the “body” element is executed in a loop as long as the condition yields “true”. Example:

```
703:<while>
704:   <condition>
705:     <expression>
           <!-- some expression which yields an boolean value -->
714:     </expression>
715:   </condition>
716:   <body>
         <!-- a sequence of statements -->
734:   </body>
735:</while>
```

3.5.7 “do-while” - Element

This element represents a do-while loop (a conditional loop where the condition is checked after each run). It contains an optional “body” element followed by a “condition” element which are used in the same way as in while loops.

3.5.8 “for” - Element

This element represents a for – loop. This kind of loop is used to count a variable from a start value to an end value. The for – element contains the elements “loop-variable”, “from”, “to”, “delta” and an optional “body” element (which contains a sequence of statements).

The “loop-variable” element contains a string which is the name of the variable. The “from”, “to”

and “delta” elements contain expressions which all have to evaluate to numerical values. The value of “delta” specifies the steps in which the variable is incremented (or decremented if “delta” contains a negative value):

```
753:<for>
754:  <loop-variable>a</loop-variable>
755:  <from>
756:    <expression>
757:      <literal type="xs:integer">0</literal>
758:    </expression>
759:  </from>
760:  <to>
761:    <expression>
762:      <literal type="xs:integer">7</literal>
763:    </expression>
764:  </to>
765:  <delta>
766:    <expression>
767:      <literal type="xs:integer">1</literal>
768:    </expression>
769:  </delta>
770:  <body>
        <!-- a sequence of statements -->
777:  </body>
778:</for>
```

3.5.9 “with” - Element

The “with” element represents the with – statement of the component pascal language. According to the Component Pascal Language Report, with – statements “execute a statement sequence depending on the result of a type test and apply a type guard to every occurrence of the tested variable within this statement sequence” [OBERON01]. I decided to implement the with – statement directly in XAST so the compilers where easy to create. It would also be possible to implement “with” statements using “if” statements and type casts but that is out of the scope of this thesis.

The “with” element contains one or more “with-block” elements. Each of those contains an optional “type-guard” element and a required “body” element. The “body” element contains a sequence of statements as described in this section (starting on page 22).

The “type-guard” element has two required child elements: A “select” element (described on page 18) which selects the variable for the type guard and an “instance-of” element which selects the type to be tested in this type guard. The “instance-of” element itself contains a “select” element:

```
822:<with>
823:  <with-block>
824:    <type-guard>
825:      <select name="x"/>
```

```
826:       <instance-of>
827:         <select name="SomeClass"/>
828:       </instance-of>
829:     </type-guard>
830:     <body>
           <!-- statements (executed if x is instance of SomeClass) -->
837:     </body>
838:   </with-block>
839:   <with-block>
840:     <type-guard>
841:       <select name="x"/>
842:       <instance-of>
843:         <select name="DerivedClass"/>
844:       </instance-of>
845:     </type-guard>
846:     <body>
           <!-- statements (executed if x is instance of DerivedClass) -->
853:     </body>
854:   </with-block>
855:   <with-block>
856:     <body>
           <!-- statements (executed in all other cases) -->
863:     </body>
864:   </with-block>
865:</with>
```

3.6 Expressions

In XAST the operands for an operator are defined by the XML tree itself, so expressions do not need special rules for the bind levels of the operators. This section shows how expressions from the Component Pascal programming language are simplified in XAST.

3.6.1 Introduction

In most programming languages the grammar for expressions is rather complicated. This is because concepts like operator-bind-levels and parenthesis have to be parsed correctly. The grammar for expressions in Component Pascal is:

```
 97:    Expr = SimpleExpr [Relation SimpleExpr].
 98:    SimpleExpr = ["+" | "-"] Term {AddOp Term}.
 99:    Term = Factor {MulOp Factor}.
100:    Factor = Designator | number | character | string | "NIL" | Set
101:             | "(" Expr ")" | " ~ " Factor.
102:    Set = "{" [Element {"," Element}] "}".
103:    Element = Expr [".." Expr].
104:    Relation = ("=" | "#" | "<" | "<=" | ">" | ">=" | "IN" | "IS").
105:    AddOp = ("+" | "-" | "OR").
106:    MulOp = ("*" | "/" | "DIV" | "MOD" | "&").
```

The compiler which was described in the chapter “Compiling Component Pascal to XML with

Coco/R" (page 9) translates this directly to the intermediate XML file format. That means that there exists an XML element for each production in this grammar. So even a really simple code like the literal String "Hello xast world" is compiled to this rather complicated XML code (which is a direct representation of the syntax tree generated by the compiler):

```
 11:<expr>
 12:  <simpleExpr>
 13:    <term>
 14:      <factor>
 15:        <literal type="xs:string">Hello xast world</literal>
 16:      </factor>
 17:    </term>
 18:  </simpleExpr>
 19:</expr>
```

As mentioned earlier in this chapter this is because the compiler has to treat the bind levels of operators correctly (multiplication operators bind stronger than addition operators which bind stronger than relation operators).

In the XML file format we do not need this separation of the operators anymore because the operands of each operator are uniquely defined by the XML code. Consider the following example where the component pascal code

```
 120:     x := x-2*y <= 7;
```

was compiled to the following XML snipplet:

```
1286:<expr operator="&lt;=">
1287:  <simpleExpr operator="-">
1288:    <term>
1289:      <factor>
1290:        <designator>
1291:          <select name="x"/>
1292:        </designator>
1293:      </factor>
1294:    </term>
1295:    <simpleExpr>
1296:      <term operator="*">
1297:        <factor>
1298:          <literal type="xs:integer">2</literal>
1299:        </factor>
1300:        <term>
1301:          <factor>
1302:            <designator>
1303:              <select name="y"/>
1304:            </designator>
1305:          </factor>
1306:        </term>
1307:      </term>
1308:    </simpleExpr>
1309:  </simpleExpr>
1310:  <simpleExpr>
```

```
1311:     <term>
1312:       <factor>
1313:         <literal type="xs:integer">7</literal>
1314:       </factor>
1315:     </term>
1316:   </simpleExpr>
1317:</expr>
```

The “*” - operator on line 1296 can only have two operands: The literal “2” on line 1298 and the variable “y” on line 1303. All other XML elements around these 2 are in fact redundant. The 2 operands for each binary operator are defined by the 2 child elements of the corresponding XML element.

In the next translation step (“Transformation of the compiler output to XAST”, page 31) the code above will be simplified to:

```
1003:<expression>
1004:   <binary-operator select="&lt;=">
1005:     <operand-1>
1006:       <binary-operator select="-">
1007:         <operand-1>
1008:           <select name="x"/>
1009:         </operand-1>
1010:         <operand-2>
1011:           <binary-operator select="*">
1012:             <operand-1>
1013:               <literal type="xs:integer">2</literal>
1014:             </operand-1>
1015:             <operand-2>
1016:               <select name="y"/>
1017:             </operand-2>
1018:           </binary-operator>
1019:         </operand-2>
1020:       </binary-operator>
1021:     </operand-1>
1022:     <operand-2>
1023:       <literal type="xs:integer">7</literal>
1024:     </operand-2>
1025:   </binary-operator>
1026:</expression>
```

In this case even the “operand-1” and “operand-2” elements would not be necessary, but they have been added for better readability of the XAST code.

3.6.2 “expression” - Element

The “expression” element contains exactly one of the following elements: “unary-operator”, “binary-operator”, “literal”, “set” or “select”. The description of the “select” - Element can be found in the section “Scopes” on page 18.

3.6.3 “unary-operator” - Element

The “unary-operator” element represents operators which have only one operand. The element has a required “select” - attribute which selects the operator to use. The “unary-operator” element contains exactly one of the following elements: “unary-operator”, “binary-operator”, “literal”, “set” or “select”.

3.6.4 “binary-operator” - Element

The “binary-operator” element has a required “select” - attribute which selects the operator to be used. The element has always two child elements: “operand-1” and “operand-2”. Both elements contain exactly one of the following elements: “unary-operator”, “binary-operator”, “literal”, “set” or “select”.

3.6.5 “literal” - Element

The “literal” element contains a text representation of the literal value. It has a required “type” - attribute which contains the XML Schema type of the literal value (i.e. “xs:integer” or “xs:string”).

3.6.6 “set” - Element

The “set” element represents sets from Component Pascal. Such sets contain an arbitrary number of “element” elements. Each such element contains either a “value” element (which contains an expression) or it contains a “range” element which contains two nested elements: “from” and “to” (both contain an expression).

4 Transformation of the compiler output to XAST

The transformation of the XML file created by the compiler described in the chapter “Compiling Component Pascal to XML with Coco/R” (page 9) to the file format described in the chapter “Abstract Syntax Trees in XML: XAST” (page 15) is done with XSLT. XSL Transformations (XSLT) is a language for transforming XML documents into other documents. XSLT is part of XSL which is a stylesheet language for XML [XSLT99]. To process the transformation any XSLT processor can be used.

The stylesheet used to transform the intermediate file format to XAST is very straight-forward in most cases, and these simple transformations are not explained here in this chapter. This chapter only describes important concepts used throughout the stylesheet and the more complex cases.

4.1 Re-writing parts of the code

There are some cases where information is added which was not explicitly present in the intermediate file format. The XSLT transformation knows about this implicit information in the intermediate language and adds information about it to the XAST file (which means it converts implicit information to explicit information). This will be exemplified in the next two subsections.

There are also some cases where explicit information from the intermediate file has to be converted to a different format. As an example for this the transformation of primitive type names will be explained later in this section.

4.1.1 Scopes

The intermediate file format contains no information about scopes because the scopes are defined by the scope rules of component pascal there. In XAST explicit scope information is needed because XAST doesn’t make any assumptions about scopes (which parts of the code has it’s own scope or how scopes are nested).

This means the stylesheet used to transform the file format has to “know” about the scope rules of component pascal and add the information to the output file. This is achieved by passing the current scope information as a template parameter to all XSLT templates where a nested scope could possibly be created.

Any template which really creates a nested scope modifies this information by appending the

current name to the scope name and then creates the according “scope” element. If for example a template got the parameter “SomeModule.global.SomeClass” as the name for the current scope and wants to create the scope for the method “fooMethod”, it would create the following new scope name (which is then passed to all called templates): “SomeModule.global.SomeClass.fooMethod”. The following code will be created:

```
0001:<scope name="SomeModule.global.SomeClass.fooMethod">
0002:   <using-scope scope="SomeModule.global.SomeClass"/>
...
0031:</scope>
```

4.1.2 Procedure return type and primitive data types

The return type of a procedure is only explicitly present in the intermediate file format if it is not void (i.e. if the procedure is not a function). In XAST there are only functions (or methods) which always have a return type. This return type has to be “void” if the function does not return any value.

The code for this uses the “xsl:choose” element to test if the procedure has a return type element and if not it adds the code for the return type “void”. There are other cases where information which was optional in the intermediate file format is required in XAST, these cases are solved similarly.

For primitive data types the type names have to be changed from the internal type names used in component pascal (like “INTEGER” or “REAL”) to the XML Schema data types used in XAST (like “xs:int” or “xs:double”). This is also done with “xsl:choose”. Other cases where values are simply transformed to a different value are solved similarly.

4.2 Processing Methods and Functions

In XAST the position of a procedure declaration in the file is not relevant, which means that it is not required that the declaration occurs before the first use of the procedure. This means also that forward declarations are not necessary, so they are simply ignored. This can be done by testing for the “forwardDeclaration” - attribute in the “procedure” element.

Anyway, the XSLT stylesheet has to distinguish between global procedures and methods. In the intermediate file format, both global procedures and methods are stored outside of record definitions, but methods have a nested “receiver” element. While processing the global procedures only “procedure” elements are taken into account which do not contain a “receiver” element.

When processing a “record” element, all procedures which belong to the record have to be processed as methods of the created “class” element. This is done with the following XSLT code:

```
256:<xsl:apply-templates select="//procedure[receiver/@type=$class-name]"
                         mode="method">
257:  <xsl:with-param name="parent-scope" select="$scope"/>
258:</xsl:apply-templates>
```

This means all “procedure” elements from the input file are processed which have a nested “receiver” element and where the “type” - attribute of this receiver element is the current class name.

4.3 Processing different statement types

The processing of the different statement types is very simple because XSLT supports different templates for the same XML element. The “match” - attribute of the template can contain any XPath expression to select the elements to be processed with the template.

For every statement type (like “assignment”) there is a template like:

```
536:<xsl:template match="statement[@type='assignment']">
...
545:</xsl:template>
```

All these templates are called from within “statementSequence” with “<xsl:**apply-templates select="statement"/>**” and the XSLT processor then selects the appropriate template for each statement element based on the XPath expression in the “match” - attribute.

4.4 Processing expressions

The XSLT template for the element “expr” from the intermediate file format has to remove the redundant parts of the expressions which have been described in the section about expressions in the previous chapter (page 27). Here the transformation has to distinguish which kind of operator is used (unary or binary) and which elements are the operands of the operator. This is done following these rules:

- If the “expr” element has an “operator” - attribute it is a binary operator and the two operands are the results of the transformation of two “simpleExpr” elements. Otherwise the result for the “expr” element is the result of the transformation of the nested “simpleExpr” element.
- If a “simpleExpr” element has a “sign” - attribute, a unary operator element has to be created for the sign and the result of the transformation of the “simpleExpr” is the operand of this unary operator. Otherwise the “simpleExpr” is processed without creating a unary operator.

- If a “simpleExpr” element has an “operator” - attribute the result is a binary operator. The first operand is the result of the transformation of the nested “term” element, the second operand is the result of the transformation of the nested “simpleExpr” element. If there is no “operator” - attribute the result of the transformation of the nested “term” element is directly used.
- If a “term” element has an “operator” - attribute the result is a binary operator. The first operand is the result of the transformation of the nested “factor” element, the second operand is the result of the transformation of the nested “term” element. If there is no “operator” - attribute the result of the transformation of the nested “factor” element is directly used.
- Within “factor” elements the code for the different types of factors (procedure calls, expressions, literals, ...) is created.

5 Conclusions

This thesis has shown that XML can be used to store the abstract syntax tree of a compiler. The straight-forward approach which just creates an XML element for every production in the grammar of a programming language is easy to realize and is used by the compiler which was described in chapter 3. This compiler can not process all aspects of component pascal correctly since some of them would require the compiler to know details about the context. A compiler which also treats this context information correctly could be built too, but it would be more complicated and was not necessary to prove the concept.

The intermediate file format does not really have an advantage compared to the source code from which it was created because it is the result of a direct transformation, it even has the disadvantage that XML parsers are normally slower than parsers for conventional programming languages. To utilize the benefits of XML we'd need a file format which is common to many programming languages. This would mean that for any programming language a compiler could create the XML representation and the same code generator could be used for all these programming language (because it would only operate on the XML representation).

The XAST file format described in chapter 3 was designed to fit these needs. In several cases it used more general concepts than the intermediate file format. It also removes redundant information like the parts in expressions which represent the bind levels of operators. The XAST file format still contains some constructs which come from the component pascal language, like the "with" element which would have been hard to express with more general constructs. Even if the XAST file format can not be used as-is as a general representation for abstract syntax trees it is a good proof-of-concept and it could be used as a basis for further work.

Bibliography

[XML04] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan: Extensible Markup Language (XML) 1.1 (2004) http://www.w3.org/TR/2004/REC-xml11-20040204/

[XSD04] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn: XML Schema Part 1: Structures Second Edition (2004) http://www.w3.org/TR/2004/REC-xmlschema-1-20041028

[XSLT99] James Clark: XSL Transformations (XSLT) Version 1.0 (1999) http://www.w3.org/TR/xslt

[WILSON03] Gregory V. Wilson: XML-Based Programming Systems(2003) Doctor Dobb's Journal March 2003

[MCGRATH03] Sean McGrath: The Merits of XML based programming languages (2003) http://expertanswercenter.techtarget.com/eac/knowledgebaseAnswer/0,295199,sid63_gci984723,00.html

[EL04] Emergent Logic LLC: SuperX++ - XML based object oriented Programming (2004) http://www.emergentlogic.com/pageXpp.php

[CM04] Clear Methods, Inc.: Water - Language for Simplified Web Services and XML Programming (2004) http://www.waterlanguage.org/

[CM03] Clear Methods, Inc: A precise and concise markup syntax that is a superset of XML (2003) http://www.concisexml.org/

[MÖSSENBÖCK05] Hanspeter Mössenböck, Albrecht Wöß, Markus Löberbauer: The Compiler Generator Coco/R (2005) http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

[PIRRINGER06] Rene Pirringer, David Tanzer: Guglhupf Commons Resource Repository (2006) http://dev.guglhupf.net/commons/rr/index.html

[MÖSSENBÖCK93] H. Mössenböck, N. Wirth: The Programming Language Oberon-2 (1993) http://www-vs.informatik.uni-ulm.de:81/projekte/Oberon-2.Report/

[OBERON01] Oberon Microsystems, Inc: Component Pascal Language Report (2001) http://www.oberon.ch/pdf/CP-Lang.pdf

Appendix A: xast.xsd

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2:
 3:<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 4:    <xs:element name="xast-module">
 5:        <xs:complexType>
 6:            <xs:sequence>
 7:                <xs:element ref="import-module" minOccurs="0"
                               maxOccurs="unbounded"/>
 8:                <xs:element ref="scope" minOccurs="1" maxOccurs="1"/>
 9:            </xs:sequence>
10:
11:            <xs:attribute ref="name" use="required"/>
12:        </xs:complexType>
13:    </xs:element>
14:
15:    <xs:element name="scope">
16:        <xs:complexType>
17:            <xs:sequence>
18:                <xs:element ref="using-scope" minOccurs="0"
                               maxOccurs="unbounded"/>
19:                <xs:element ref="constant" minOccurs="0"
                               maxOccurs="unbounded"/>
20:                <xs:element ref="type" minOccurs="0"
                               maxOccurs="unbounded"/>
21:                <xs:element ref="variable" minOccurs="0"
                               maxOccurs="unbounded"/>
22:
23:                <xs:element ref="function" minOccurs="0"
                               maxOccurs="unbounded"/>
24:                <xs:element ref="method" minOccurs="0"
                               maxOccurs="unbounded"/>
25:
26:                <xs:element ref="parameters" minOccurs="0"
                               maxOccurs="1"/>
27:                <xs:element ref="return-type" minOccurs="0"
                               maxOccurs="1"/>
28:                <xs:element ref="this-pointer" minOccurs="0"
                               maxOccurs="1"/>
29:                <xs:element ref="body" minOccurs="0" maxOccurs="1"/>
30:
31:                <xs:element ref="module-initializer" minOccurs="0"
                               maxOccurs="1"/>
32:                <xs:element ref="module-finalizer" minOccurs="0"
                               maxOccurs="1"/>
33:            </xs:sequence>
34:
35:            <xs:attribute ref="name" use="required"/>
36:        </xs:complexType>
37:    </xs:element>
38:
39:    <xs:element name="import-module">
40:        <xs:complexType>
41:            <xs:attribute ref="name" use="required"/>
42:        </xs:complexType>
43:    </xs:element>
```

```
 44:
 45:     <xs:element name="using-scope">
 46:         <xs:complexType>
 47:             <xs:attribute ref="scope" use="required"/>
 48:             <xs:attribute ref="prefix" use="optional"/>
 49:         </xs:complexType>
 50:     </xs:element>
 51:
 52:     <xs:element name="constant">
 53:         <xs:complexType>
 54:             <xs:sequence>
 55:                 <xs:element ref="value" minOccurs="1" maxOccurs="1"/>
 56:             </xs:sequence>
 57:
 58:             <xs:attribute ref="name" use="required"/>
 59:             <xs:attribute ref="access" use="required"/>
 60:             <xs:attribute ref="read-only" use="required"/>
 61:         </xs:complexType>
 62:     </xs:element>
 63:
 64:     <xs:complexType name="value-type">
 65:         <xs:choice>
 66:             <xs:element ref="expression"/>
 67:         </xs:choice>
 68:     </xs:complexType>
 69:
 70:     <xs:element name="value" type="value-type"/>
 71:
 72:     <xs:element name="variable">
 73:         <xs:complexType>
 74:             <xs:sequence>
 75:                 <xs:element ref="type" minOccurs="1" maxOccurs="1"/>
 76:             </xs:sequence>
 77:
 78:             <xs:attribute ref="name" use="required"/>
 79:             <xs:attribute ref="access" use="required"/>
 80:             <xs:attribute ref="read-only" use="required"/>
 81:         </xs:complexType>
 82:     </xs:element>
 83:
 84:     <xs:element name="function">
 85:         <xs:complexType>
 86:             <xs:sequence>
 87:                 <xs:element ref="scope"/>
 88:             </xs:sequence>
 89:
 90:             <xs:attribute ref="name" use="required"/>
 91:             <xs:attribute ref="access" use="required"/>
 92:             <xs:attribute ref="read-only" use="required"/>
 93:         </xs:complexType>
 94:     </xs:element>
 95:
 96:     <xs:element name="method">
 97:         <xs:complexType>
 98:             <xs:sequence>
 99:                 <xs:element ref="scope"/>
100:             </xs:sequence>
101:
```

```
102:             <xs:attribute ref="name" use="required"/>
103:             <xs:attribute ref="access" use="required"/>
104:             <xs:attribute ref="read-only" use="required"/>
105:             <xs:attribute ref="new" use="optional"/>
106:             <xs:attribute ref="empty" use="optional"/>
107:             <xs:attribute ref="abstract" use="optional"/>
108:             <xs:attribute ref="extensible" use="optional"/>
109:         </xs:complexType>
110:     </xs:element>
111:
112:     <xs:element name="module-initializer">
113:         <xs:complexType>
114:             <xs:sequence>
115:                 <xs:element ref="scope" minOccurs="1" maxOccurs="1"/>
116:             </xs:sequence>
117:         </xs:complexType>
118:     </xs:element>
119:
120:     <xs:element name="module-finalizer">
121:         <xs:complexType>
122:             <xs:sequence>
123:                 <xs:element ref="scope" minOccurs="1" maxOccurs="1"/>
124:             </xs:sequence>
125:         </xs:complexType>
126:     </xs:element>
127:
128:     <xs:complexType name="all-types">
129:         <xs:choice>
130:             <xs:element ref="simple-type"/>
131:             <xs:element ref="array-type"/>
132:             <xs:element ref="pointer-type"/>
133:             <xs:element ref="class"/>
134:             <xs:element ref="function-type"/>
135:         </xs:choice>
136:         <xs:attribute ref="name" use="optional"/>
137:         <xs:attribute ref="access" use="optional"/>
138:         <xs:attribute ref="read-only" use="optional"/>
139:     </xs:complexType>
140:
141:     <xs:element name="type" type="all-types"/>
142:
143:     <xs:element name="simple-type">
144:         <xs:complexType>
145:             <xs:sequence>
146:                 <xs:element ref="select" minOccurs="1" maxOccurs="1"/>
147:             </xs:sequence>
148:         </xs:complexType>
149:     </xs:element>
150:
151:     <xs:element name="pointer-type" type="all-types"/>
152:
153:     <!-- NOTE Multiple inheritance is possible in XAST since it is
154:          inteded to be a multi-purpose format for syntax trees.
155:     -->
156:     <xs:element name="class">
157:         <xs:complexType>
158:             <xs:sequence>
159:                 <xs:element ref="baseclass" minOccurs="0"
```

```
                            maxOccurs="unbounded"/>
160:                <xs:element ref="scope" minOccurs="1" maxOccurs="1"/>
161:            </xs:sequence>
162:        </xs:complexType>
163:    </xs:element>
164:
165:    <xs:element name="baseclass">
166:        <xs:complexType>
167:            <xs:sequence>
168:                <xs:element ref="select" minOccurs="1" maxOccurs="1"/>
169:            </xs:sequence>
170:        </xs:complexType>
171:    </xs:element>
172:
173:    <xs:element name="function-type">
174:        <xs:complexType>
175:            <xs:sequence>
176:                <xs:element ref="parameters" minOccurs="1"
                            maxOccurs="1"/>
177:                <xs:element ref="return-type" minOccurs="1"
                            maxOccurs="1"/>
178:            </xs:sequence>
179:        </xs:complexType>
180:    </xs:element>
181:
182:    <xs:element name="parameters">
183:        <xs:complexType>
184:            <xs:sequence>
185:                <xs:element ref="parameter" minOccurs="0"
                            maxOccurs="unbounded"/>
186:            </xs:sequence>
187:        </xs:complexType>
188:    </xs:element>
189:
190:    <xs:element name="parameter">
191:        <xs:complexType>
192:            <xs:choice>
193:                <xs:element ref="type"/>
194:                <xs:element ref="expression"/>
195:            </xs:choice>
196:
197:            <xs:attribute ref="parameter-type" use="optional"/>
198:            <xs:attribute ref="name" use="optional"/>
199:        </xs:complexType>
200:    </xs:element>
201:
202:    <xs:element name="return-type" type="all-types"/>
203:
204:    <xs:element name="this-pointer">
205:        <xs:complexType>
206:            <xs:attribute ref="name"/>
207:        </xs:complexType>
208:    </xs:element>
209:
210:    <xs:complexType name="all-statements">
211:        <xs:sequence>
212:            <!-- declarations... -->
213:            <xs:element ref="constant" minOccurs="0"
```

```
                        maxOccurs="unbounded"/>
214:            <xs:element ref="type" minOccurs="0" maxOccurs="unbounded"/>
215:            <xs:element ref="variable" minOccurs="0"
                        maxOccurs="unbounded"/>
216:            <xs:element ref="function" minOccurs="0"
                        maxOccurs="unbounded"/>
217:
218:            <!-- statements -->
219:            <xs:choice minOccurs="0" maxOccurs="unbounded">
220:                <xs:element ref="assignment"/>
221:                <xs:element ref="call"/>
222:                <xs:element ref="if"/>
223:                <xs:element ref="switch"/>
224:                <xs:element ref="while"/>
225:                <xs:element ref="do-while"/>
226:                <xs:element ref="for"/>
227:                <xs:element ref="with"/>
228:                <xs:element name="exit">
229:                    <xs:complexType/>
230:                </xs:element>
231:                <xs:element name="return">
232:                <xs:complexType>
233:                    <xs:choice minOccurs="0" maxOccurs="1">
234:                        <xs:element ref="expression"/>
235:                    </xs:choice>
236:                </xs:complexType>
237:                </xs:element>
238:            </xs:choice>
239:        </xs:sequence>
240:    </xs:complexType>
241:
242:    <xs:element name="body" type="all-statements"/>
243:
244:    <xs:element name="assignment">
245:        <xs:complexType>
246:            <xs:sequence>
247:                <xs:element ref="target" minOccurs="1" maxOccurs="1"/>
248:                <xs:element ref="value" minOccurs="1" maxOccurs="1"/>
249:            </xs:sequence>
250:        </xs:complexType>
251:    </xs:element>
252:
253:    <xs:element name="target">
254:        <xs:complexType>
255:            <xs:sequence>
256:                <xs:element ref="select" minOccurs="1" maxOccurs="1"/>
257:            </xs:sequence>
258:        </xs:complexType>
259:    </xs:element>
260:
261:    <xs:element name="call">
262:        <xs:complexType>
263:            <xs:sequence>
264:                <xs:element name="method">
265:                    <xs:complexType>
266:                        <xs:sequence>
267:                            <xs:element ref="select" minOccurs="1"
                                        maxOccurs="1"/>
```

```
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element ref="parameters"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="if">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="condition" minOccurs="1"
                            maxOccurs="1"/>
                <xs:element ref="then" minOccurs="1" maxOccurs="1"/>
                <xs:element ref="else-if" minOccurs="0"
                            maxOccurs="unbounded"/>
                <xs:element ref="else" minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="condition" type="value-type"/>
    <xs:element name="then" type="all-statements"/>
    <xs:element name="else" type="all-statements"/>

    <xs:element name="else-if">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="condition" minOccurs="1"
                            maxOccurs="1"/>
                <xs:element ref="then" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!-- I guess it would not make sense to prohibit switch-statements
         with no cases, so they are allowed. The same applies for empty
         cases (without a body - element) - see below.
    -->
    <xs:element name="switch">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="condition" minOccurs="1"
                            maxOccurs="1"/>
                <xs:element ref="case" minOccurs="0"
                            maxOccurs="unbounded"/>
                <xs:element ref="default" minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="case">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="label" minOccurs="1"
                            maxOccurs="unbounded"/>
                <xs:element ref="body" minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
```

```
320:         </xs:complexType>
321:     </xs:element>
322:
323:     <xs:element name="default" type="all-statements"/>
324:
325:     <xs:element name="label">
326:         <xs:complexType>
327:             <xs:choice>
328:                 <xs:element ref="value"/>
329:                 <xs:element ref="range"/>
330:             </xs:choice>
331:         </xs:complexType>
332:     </xs:element>
333:
334:     <xs:element name="range">
335:         <xs:complexType>
336:             <xs:sequence>
337:                 <xs:element name="from" minOccurs="1" maxOccurs="1"
                                type="value-type"/>
338:                 <xs:element name="to" minOccurs="1" maxOccurs="1"
                                type="value-type"/>
339:             </xs:sequence>
340:         </xs:complexType>
341:     </xs:element>
342:
343:     <xs:element name="while">
344:         <xs:complexType>
345:             <xs:sequence>
346:                 <xs:element ref="condition" minOccurs="1"
                                maxOccurs="1"/>
347:                 <xs:element ref="body" minOccurs="0" maxOccurs="1"/>
348:             </xs:sequence>
349:         </xs:complexType>
350:     </xs:element>
351:
352:     <xs:element name="do-while">
353:         <xs:complexType>
354:             <xs:sequence>
355:                 <xs:element ref="body" minOccurs="0" maxOccurs="1"/>
356:                 <xs:element ref="condition" minOccurs="1"
                                maxOccurs="1"/>
357:             </xs:sequence>
358:         </xs:complexType>
359:     </xs:element>
360:
361:     <xs:element name="for">
362:         <xs:complexType>
363:             <xs:sequence>
364:                 <xs:element name="loop-variable" type="xs:string"
                                minOccurs="1" maxOccurs="1"/>
365:                 <xs:element name="from" type="value-type" minOccurs="1"
                                maxOccurs="1"/>
366:                 <xs:element name="to" type="value-type" minOccurs="1"
                                maxOccurs="1"/>
367:                 <xs:element name="delta" type="value-type" minOccurs="1"
                                maxOccurs="1"/>
368:                 <xs:element ref="body"  minOccurs="0" maxOccurs="1"/>
369:             </xs:sequence>
```

```
370:         </xs:complexType>
371:     </xs:element>
372:
373:     <xs:element name="with">
374:         <xs:complexType>
375:             <xs:sequence>
376:                 <xs:element ref="with-block" minOccurs="1"
                                 maxOccurs="unbounded"/>
377:             </xs:sequence>
378:         </xs:complexType>
379:     </xs:element>
380:
381:     <xs:element name="with-block">
382:         <xs:complexType>
383:             <xs:sequence>
384:                 <xs:element ref="type-guard" minOccurs="0"
                                 maxOccurs="1"/>
385:                 <xs:element ref="body" minOccurs="1" maxOccurs="1"/>
386:             </xs:sequence>
387:         </xs:complexType>
388:     </xs:element>
389:
390:     <xs:element name="type-guard">
391:         <xs:complexType>
392:             <xs:sequence>
393:                 <xs:element ref="select" minOccurs="1" maxOccurs="1"/>
394:                 <xs:element ref="instance-of" minOccurs="1"
                                 maxOccurs="1"/>
395:             </xs:sequence>
396:         </xs:complexType>
397:     </xs:element>
398:
399:     <xs:element name="instance-of">
400:         <xs:complexType>
401:             <xs:sequence>
402:                 <xs:element ref="select" minOccurs="1" maxOccurs="1"/>
403:             </xs:sequence>
404:         </xs:complexType>
405:     </xs:element>
406:
407:     <xs:element name="array-type">
408:         <xs:complexType>
409:             <xs:sequence>
410:                 <xs:element ref="length" minOccurs="1" maxOccurs="1"/>
411:                 <xs:element ref="element-type" minOccurs="1"
                                 maxOccurs="1"/>
412:             </xs:sequence>
413:         </xs:complexType>
414:     </xs:element>
415:
416:     <xs:element name="length">
417:         <xs:complexType>
418:             <xs:choice minOccurs="0" maxOccurs="1">
419:                 <xs:element ref="expression"/>
420:             </xs:choice>
421:         </xs:complexType>
422:     </xs:element>
423:
```

```
424:     <xs:element name="element-type" type="all-types"/>
425:
426:     <xs:element name="select">
427:         <xs:complexType>
428:             <xs:sequence>
429:                 <xs:element ref="select" minOccurs="0" maxOccurs="1"/>
430:             </xs:sequence>
431:
432:             <xs:attribute ref="name" use="required"/>
433:         </xs:complexType>
434:     </xs:element>
435:
436:     <xs:element name="literal">
437:         <xs:complexType mixed="true">
438:             <xs:attribute ref="type" use="required"/>
439:         </xs:complexType>
440:     </xs:element>
441:
442:     <xs:element name="expression">
443:         <xs:complexType>
444:             <xs:choice>
445:                 <xs:element ref="unary-operator"/>
446:                 <xs:element ref="binary-operator"/>
447:                 <xs:element ref="literal"/>
448:                 <xs:element ref="set"/>
449:                 <xs:element ref="select"/>
450:                 <xs:element ref="call"/>
451:             </xs:choice>
452:         </xs:complexType>
453:     </xs:element>
454:
455:     <xs:element name="unary-operator">
456:         <xs:complexType>
457:             <xs:choice>
458:                 <xs:element ref="unary-operator"/>
459:                 <xs:element ref="binary-operator"/>
460:                 <xs:element ref="literal"/>
461:                 <xs:element ref="set"/>
462:                 <xs:element ref="select"/>
463:                 <xs:element ref="call"/>
464:             </xs:choice>
465:
466:             <xs:attribute ref="select" use="required"/>
467:         </xs:complexType>
468:     </xs:element>
469:
470:     <xs:element name="binary-operator">
471:         <xs:complexType>
472:             <xs:sequence>
473:                 <xs:element ref="operand-1" minOccurs="1"
                                maxOccurs="1"/>
474:                 <xs:element ref="operand-2" minOccurs="1"
                                maxOccurs="1"/>
475:             </xs:sequence>
476:
477:             <xs:attribute ref="select" use="required"/>
478:         </xs:complexType>
479:     </xs:element>
```

```
480:
481:    <xs:complexType name="operand-type">
482:        <xs:choice>
483:            <xs:element ref="unary-operator"/>
484:            <xs:element ref="binary-operator"/>
485:            <xs:element ref="literal"/>
486:            <xs:element ref="set"/>
487:            <xs:element ref="select"/>
488:            <xs:element ref="call"/>
489:        </xs:choice>
490:    </xs:complexType>
491:
492:    <xs:element name="operand-1" type="operand-type"/>
493:    <xs:element name="operand-2" type="operand-type"/>
494:
495:    <xs:element name="set">
496:        <xs:complexType>
497:            <xs:sequence>
498:                <xs:element ref="element" minOccurs="0"
                                maxOccurs="unbounded"/>
499:            </xs:sequence>
500:        </xs:complexType>
501:    </xs:element>
502:
503:    <xs:element name="element">
504:        <xs:complexType>
505:            <xs:choice>
506:                <xs:element ref="value"/>
507:                <xs:element ref="range"/>
508:            </xs:choice>
509:        </xs:complexType>
510:    </xs:element>
511:
512:    <xs:attribute name="name" type="xs:string"/>
513:    <xs:attribute name="scope" type="xs:string"/>
514:    <xs:attribute name="prefix" type="xs:string"/>
515:    <xs:attribute name="access">
516:        <xs:simpleType>
517:            <xs:restriction base="xs:string">
518:                <xs:enumeration value="public"/>
519:                <xs:enumeration value="package"/>
520:                <xs:enumeration value="protected"/>
521:                <xs:enumeration value="private"/>
522:            </xs:restriction>
523:        </xs:simpleType>
524:    </xs:attribute>
525:    <!-- read-only is not the same as constant - it comes from component
526:         pascal, where it's possible to export a variable read-only (so
527:         it's a normal variable in the module/class/... where it is
528:         defined, and read-only everywhere else.
529:    -->
530:    <xs:attribute name="read-only" type="xs:boolean"/>
531:    <xs:attribute name="type" type="xs:string"/>
532:    <xs:attribute name="parameter-type">
533:        <xs:simpleType>
534:            <xs:restriction base="xs:string">
535:                <xs:enumeration value="input"/>
536:                <xs:enumeration value="output"/>
```

```
537:                <xs:enumeration value="input-output"/>
538:             </xs:restriction>
539:          </xs:simpleType>
540:       </xs:attribute>
541:       <xs:attribute name="new" type="xs:boolean"/>
542:       <xs:attribute name="empty" type="xs:boolean"/>
543:       <xs:attribute name="abstract" type="xs:boolean"/>
544:       <xs:attribute name="extensible" type="xs:boolean"/>
545:       <xs:attribute name="select" type="xs:string"/>
546:</xs:schema>
```

Appendix B: Grammar of Component Pascal

```
  1:/*
  2:Component Pascal (Compiler for creating "Abstract XML Syntax Trees"
    (AST).
  3: */
  4:COMPILER ComponentPascal
  5:
  6:CHARACTERS
  7:    /* I left out the norwegian letters here. They should be added in a
          later version.
  8:     */
  9:    letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
 10:    digit = "0123456789".
 11:    cr        = '\r'.
 12:    lf        = '\n'.
 13:    tab       = '\t'.
 14:    hexDigit = "0123456789ABCDEF".
 15:    /* I'm not sure if ANY is the right definition here, but the
          Component Pascal language report does not define
 16:     * "char" and it's only used in strings.
 17:     */
 18:    char      = ANY.
 19:
 20:IGNORE cr + lf + tab
 21:
 22:TOKENS
 23:    ident = (letter | "_") {letter | "_" | digit}.
 24:    integer = digit {digit} | digit {hexDigit} ( "H" | "L" ).
 25:    /* ScaleFactor = "E" ["+" | "-"] digit {digit}. */
 26:    real = digit {digit} "." {digit} ["E" ["+" | "-"] digit {digit}].
 27:
 28:    character = digit {hexDigit} "X".
 29:    string = '"' {char} '"' | "'" {char} "'".
 30:
 31:COMMENTS FROM "(*" TO "*)" NESTED
 32:
 33:PRODUCTIONS
 34:    number = integer | real.
 35:
 36:    ComponentPascal = Module.
 37:    Module = "MODULE" ident ";" [ImportList] DeclSeq ["BEGIN"
              StatementSeq]
 38:             ["CLOSE" StatementSeq] "END" ident ".".
 39:    ImportList = "IMPORT" [ident ":="] ident {"," [ident ":="] ident}
```

```
                  ";".
40:     DeclSeq = { "CONST" {ConstDecl ";" } | "TYPE" {TypeDecl ";"}
41:               | "VAR" {VarDecl ";"}}
42:               {ProcDecl ";" | ForwardDecl ";"}.
43:     ConstDecl = IdentDef "=" ConstExpr.
44:     TypeDecl = IdentDef "=" Type.
45:     VarDecl = IdentList ":" Type.
46:     ProcDecl = "PROCEDURE" [Receiver] IdentDef [FormalPars]
47:                ["," "NEW"] ["," ("ABSTRACT" | "EMPTY" | "EXTENSIBLE")]
48:                [";" DeclSeq ["BEGIN" StatementSeq] "END" ident].
49:     ForwardDecl = "PROCEDURE" "^" [Receiver] IdentDef [FormalPars].
50:     FormalPars = "(" [FPSection {";" FPSection}] ")" [":" Type].
51:     FPSection = ["VAR" | "IN" | "OUT"] ident {"," ident} ":" Type.
52:     Receiver = "(" ["VAR" | "IN"] ident ":" ident ")".
53:     Type = Qualident
54:            | "ARRAY" [ConstExpr {"," ConstExpr}] "OF" Type
55:            | ["ABSTRACT" | "EXTENSIBLE" | "LIMITED"] "RECORD"
56:              ["("Qualident")"] FieldList {";" FieldList} "END"
57:            | "POINTER" "TO" Type
58:            | "PROCEDURE" [FormalPars].
59:     FieldList = [IdentList ":" Type].
60:     StatementSeq = Statement {";" Statement}.
61:     Statement = [ Designator ":=" Expr
62:                 | Designator ["(" [ExprList] ")"]
63:                 | "IF" Expr "THEN" StatementSeq
64:                   {"ELSIF" Expr "THEN" StatementSeq}
65:                   ["ELSE" StatementSeq] "END"
66:                 | "CASE" Expr "OF" Case {"|" Case} ["ELSE" StatementSeq]
                      "END"
67:                 | "WHILE" Expr "DO" StatementSeq "END"
68:                 | "REPEAT" StatementSeq "UNTIL" Expr
69:                 | "FOR" ident ":=" Expr "TO" Expr ["BY" ConstExpr]
70:                   "DO" StatementSeq "END"
71:                 | "LOOP" StatementSeq "END"
72:                 | "WITH" Guard "DO" StatementSeq
73:                   {"|" Guard "DO" StatementSeq} ["ELSE" StatementSeq]
                      "END"
74:                 | "EXIT" | "RETURN" [Expr] ].
75:     Case = [CaseLabels {"," CaseLabels} ":" StatementSeq].
76:     CaseLabels = ConstExpr [".." ConstExpr].
77:     Guard = Qualident ":" Qualident.
78:     ConstExpr = Expr.
79:     Expr = SimpleExpr [Relation SimpleExpr].
80:     SimpleExpr = ["+" | "-"] Term {AddOp Term}.
81:     Term = Factor {MulOp Factor}.
82:     Factor = Designator | number | character | string | "NIL" | Set
83:              | "(" Expr ")" | " ~ " Factor.
84:     Set = "{" [Element {"," Element}] "}".
85:     Element = Expr [".." Expr].
86:     Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | "IN" | "IS".
87:     AddOp = "+" | "-" | "OR".
88:     MulOp = "*" | "/" | "DIV" | "MOD" | "&".
89:     Designator = Qualident {"." ident | "[" ExprList "]" | " ^ " | "$"
90:                  | "(" Qualident ")" | "(" [ExprList] ")"}.
91:     ExprList = Expr {"," Expr}.
92:     IdentList = IdentDef {"," IdentDef}.
93:     Qualident = [ident "."] ident.
94:     IdentDef = ident ["*" | "-"].
```

```
 95:
 96:END ComponentPascal.
```

Appendix C: Re-written Grammar for Coco/R

```
  1:/*
  2: * Component Pascal (Compiler for creating "Abstract XML Syntax Trees"
       (AST).
  3: */
  4:COMPILER ComponentPascal
  5:
  6:CHARACTERS
  7:    /* I left out the norwegian letters here. They should be added in a
         later
  8:     * version. I do not think they are nessessary to prove the concept.
  9:     */
 10:    letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
 11:    digit = "0123456789".
 12:    cr        = '\r'.
 13:    lf        = '\n'.
 14:    tab       = '\t'.
 15:    hexDigit = "0123456789ABCDEF".
 16:    /* I'm not sure if ANY is the right definition here, but the
         Component
 17:     * Pascal language report does not define "char" and it's only used
         in strings.
 18:     */
 19:    char      = ANY-'"'-"'".
 20:
 21:IGNORE cr + lf + tab
 22:
 23:TOKENS
 24:    ident = (letter | "_") {letter | "_" | digit}.
 25:    integer = digit {digit} | digit {hexDigit} ( "H" | "L" ).
 26:    /* ScaleFactor = "E" ["+" | "-"] digit {digit}. */
 27:    real = digit {digit} "." {digit} ["E" ["+" | "-"] digit {digit}].
 28:
 29:    character = digit {hexDigit} "X".
 30:    string = '"' {char} '"' | "'" {char} "'".
 31:
 32:COMMENTS FROM "(*" TO "*)" NESTED
 33:
 34:PRODUCTIONS
 35:    number = integer | real.
 36:
 37:    ComponentPascal = Module.
 38:    Module = "MODULE" ident ";" [ImportList] DeclSeq ["BEGIN"
                 StatementSeq]
 39:             ["CLOSE" StatementSeq] "END" ident ".".
 40:    /* From the Component Pascal Language Report:
 41:     * WAS: ImportList = "IMPORT" [ident ":="] ident {"," [ident ":="]
         ident} ";".
 42:     */
 43:    ImportList = "IMPORT" ident [ ":=" ident] {"," ident [ ":=" ident]}
                     ";".
 44:    DeclSeq = { "CONST" {ConstDecl ";" } | "TYPE" {TypeDecl ";"}
 45:              | "VAR" {VarDecl ";"}}
 46:              {Procedures ";"}.
```

```
47:     Procedures = "PROCEDURE" ( ProcDecl | ForwardDecl).
48:     ConstDecl = IdentDef "=" ConstExpr.
49:     TypeDecl = IdentDef "=" Type.
50:     VarDecl = IdentList ":" Type.
51:     ProcDecl = [Receiver] IdentDef [FormalPars] ["," (("NEW" ["," 
                   ("ABSTRACT"
52:                   | "EMPTY" | "EXTENSIBLE")])
53:                   | (("ABSTRACT" | "EMPTY" | "EXTENSIBLE"))))]
54:                   [";" DeclSeq ["BEGIN" StatementSeq] "END" ident].
55:     ProcDecl = [Receiver] IdentDef [FormalPars] ("," MethodAttributes
56:                   | ProcedureBody).
57:     MethodAttributes = "NEW" NewAttribute | InheritanceAttributes.
58:     NewAttribute = ProcedureBody | "," InheritanceAttributes.
59:     InheritanceAttributes = "ABSTRACT" | "EMPTY" | "EXTENSIBLE"
                                ProcedureBody.
60:     ProcedureBody = ";" DeclSeq ["BEGIN" StatementSeq] "END" ident.
61:     ForwardDecl = "^" [Receiver] IdentDef [FormalPars] ["," (("NEW"
62:                   ["," ("ABSTRACT" | "EMPTY" | "EXTENSIBLE")])
63:                   | (("ABSTRACT" | "EMPTY" | "EXTENSIBLE")))].
64:     FormalPars = "(" [FPSection {";" FPSection}] ")" [":" Type].
65:     FPSection = ["VAR" | "IN" | "OUT"] ident {"," ident} ":" Type.
66:     Receiver = "(" ["VAR" | "IN" ] ident ":" ident ")".
67:     Type = Qualident
68:            | "ARRAY" [ConstExpr {"," ConstExpr}] "OF" Type
69:            | ["ABSTRACT" | "EXTENSIBLE" | "LIMITED"] "RECORD" ["("
                 Qualident ")"]
70:              FieldList {";" FieldList } "END"
71:            | "POINTER" "TO" Type
72:            | "PROCEDURE" [FormalPars].
73:     FieldList = [IdentList ":" Type].
74:     StatementSeq = Statement {";" Statement}.
75:     Statement = [
76:                 Designator DesignatedStmt
77:                 | "IF" Expr "THEN" StatementSeq {"ELSIF" Expr "THEN"
                      StatementSeq}
78:                   ["ELSE" StatementSeq ] "END"
79:                 | "CASE" Expr "OF" Case {"|" Case} ["ELSE" StatementSeq]
                      "END"
80:                 | "WHILE" Expr "DO" StatementSeq "END"
81:                 | "REPEAT" StatementSeq "UNTIL" Expr
82:                 | "FOR" ident ":=" Expr "TO" Expr ["BY" ConstExpr] "DO"
83:                   StatementSeq "END"
84:                 | "LOOP" StatementSeq "END"
85:                 | "WITH" Guard "DO" StatementSeq
86:                   {"|" Guard "DO" StatementSeq} ["ELSE" StatementSeq]
                      "END"
87:                 | "EXIT"
88:                 | "RETURN" [Expr]
89:                 ].
90:     DesignatedStmt = (":=" Expr)
91:                      /* WAS ORIGINALLY: (["(" [ExprList] ")"]) */
92:                      | ([ActualParams]).
93:     Case = [CaseLabels {"," CaseLabels} ":" StatementSeq].
94:     CaseLabels = ConstExpr [".." ConstExpr].
95:     Guard = Qualident ":" Qualident.
96:     ConstExpr = Expr.
97:     Expr = SimpleExpr [Relation SimpleExpr].
98:     SimpleExpr = ["+" | "-"] Term {AddOp Term}.
```

```
 99:     Term = Factor {MulOp Factor}.
100:     Factor = Designator | number | character | string | "NIL" | Set
101:              | "(" Expr ")" | " ~ " Factor.
102:     Set = "{" [Element {"," Element}] "}".
103:     Element = Expr [".." Expr].
104:     Relation = ("=" | "#" | "<" | "<=" | ">" | ">=" | "IN" | "IS").
105:     AddOp = ("+" | "-" | "OR").
106:     MulOp = ("*" | "/" | "DIV" | "MOD" | "&").
107:
108:     /* The grammar according to the Component Pascal Language Report
            would be:
109:      * Designator = Qualident {"." ident | "[" ExprList "]" | " ^ " |
            "$" | "(" Qualident ")" | ActualParams}.
110:      * ActualParams = "(" [ExprList] ")".
111:      */
112:     Designator = Qualident {"." ident | "[" ExprList "]" | "^" | "$" |
                     ActualParams}.
113:     ActualParams = "(" [ExprList] ")".
114:     ExprList = Expr {"," Expr}.
115:     IdentList = IdentDef {"," IdentDef}.
116:     /* Was originally: Qualident=[ident "."] ident. */
117:     Qualident = (ident | "NEW") ["." (ident | "NEW")].
118:     IdentDef = ident [("*" | "-")].
119:
120:END ComponentPascal.
```