Directory Attribute Grammars

Alberto Manuel Simões[1] **, José João Almeida**[1] **, Pedro Rangel Henriques**[1]

[1]Departamento de Informática – Universidadeo do Minho
campus de Gualtar – 4710 Braga, Portugal

albie@alfarrabio.di.uminho.pr, jj@di.uminho.pt, prh@di.uminho.pt

Abstract. *To publish documents we must take care of documents' structure and their formal layout. If a large document is composed by a lot of parts including text, images, code and so on, they will lay in different files; so file organization should also be a concern in the mind of the publisher. Normally, we focus on the structure and design, and forget file organization until this becomes a trouble, at production stage.*
The most common way to organize files is using the file system; directory structure is very flexible but quickly becomes disorganized.
In this paper we present a way to turn a file system tree into an organized web page, using a grammatical formalism (hereafter called Directory Attribute Grammars: DAG = AG + Makefiles).
We developed a tool (similar to a compiler) to publish different formats on the web, automatically, from the file structure and a DAG specification. This tool allows us to maintain a huge number of documents of different types, and process them systematically.

1. Introduction

This paper is concerned with Directory Attribute Grammars (DAG[1]) aimed at the systematic publishing of documents, as web pages and other electronic formats, from files organized in a file system tree.

We can see DAG as a Domain Specific Language[van Deursen et al., 2000, Lämmel and Mernik, 2001] to describe directories' structure and the way we can process each file in the tree. It is a specification language designed specially for that purpose. On one hand, allows to describe the file system structure and the file processing in a very easy way and, on other hand embed an engine to process the files in a way very similar to the Linux makefiles.

The motivation for this work arose from a heavy task we were faced with some time ago: we needed to organize and publish three hundred XML[Consortium, 1998] documents of different kinds and related image documents (more than five hundred).

The documents to be processed were already stored in the filesystem, organized by directories accordingly with their origin. The task we should implement was, roughly speaking, a set of translations from XML to HTML.

[1]We know that DAG usually stands for Directed Acyclic Graphs. Forget that meaning while reading our ideas.

We would like, not only to make possible the navigation over the archive, but also to publish the documents in different formats: HTML, Postscript, etc (we should ensure an easy way to produce more formats when needed). These other format can be of any kind, which lead us to produce a dynamic catalog whenever the document repository is changed.

At a first stage, we used the file system as a storing facility because we needed to store the information and did not have the time not the hardware to set up a database tool. This way, the file system was the most simple way to store the information, because:

- it is an easy and natural way to store heterogeneous information;
- information is kept in text format, independent of the operating system or tools;
- it is possible to use operating system commands and tools (like CVS, makefiles, converting utilities, scripting languages, etc.) to maintain and process the document files; For example, we keep the file system tree under CVS making it possible to checkout, at any moment, a recent copy and update in little time the web-site; it makes possible to correct and edit documents in a concurrent way which is crucial for cooperative work;
- it is possible to use file system attributes (like time stamps, etc.) to manage in a smart way the document collection (for instance, avoiding to rebuild unnecessary documents);

However this approach has a drawback: the file system quickly can become chaotic and disorganized. This was the reason why we built the DAG tool, to process the documents tree and to check the file system structure. It main goals are:

- to define a formalism to specify what can be found in the file-system (semantic structure) to avoid chaos;
- to include in that declarative formalism a description of the file-system transformations (to build different views of the given documents, to establish relations and links between them, or to extract other documents from the initial ones);
- to develop a tool to interpret the previous specification, executing the desired transformations.

The above referred transformations should be thought of as a structural processing of the file-system. Moreover, we do not want to write different processing rules for all particular element of the file-system. Each rule should affect all elements of the same type.

While the attribute grammar formalism is good to define our semantic structure and transformation goals, we want to include some of makefiles' concepts, mainly the dependencies strategy and the timestamps to avoid unnecessary regeneration of files (some sort of file system incremental evaluation).

In order to assure an easy and complete access to the set of tools available in the operating system, we want do express the transformation rules in a scripting language[2].

In this paper we will present the Directory Attribute Grammars formalism (section 2.), followed by a detailed example (section 3.). Section 4. is devoted to the Directory

[2]In the current version, perl[Christiansen and Torkington, 1999, Wall et al.,] was chosen.

Attribute Grammars processor. In section 5. we describe an application of DAG and its processor to a real size case and we show some measures to assess the performance of the proposed tool.

2. Approaching File System processing with Grammars

An Attribute Grammar [Knuth, 1968, Waite and Goos, 1984, Waite and Carter, 1993, Deransart and Jourdan, 1990, Alblas and Melichar, 1991] (AG for short) is a well accepted formalism, in the area of compiler development and language processing, to specify the syntax and static semantics of (programming) languages. AG is a declarative formalism, supported on a traditional Context Free Grammar (CFG), that can be extended to couple also with the dynamic semantics, allowing for the specification of the translator.

An AG is just a standard notation to describe a language and its transformation, but it is well known that a program to process sentences of that language can the derived systematically from the specification (the AG). This systematic construction can be done automatically by a program that is called a *compiler generator*[3](CG for short) [Johnson, 1975, Dencker, 1980, Kastens et al., 1982, Kastens, 1991, Parr et al., 1993, Gray et al., 1990, Kasters, 1994, Sloam, 1994, Saraiva, 2000, Mernik et al., 2000].

A processor developed according to an AG will be called a *semantics-directed translator*, in opposition to the traditional *syntax-directed translator* based on a CFG with semantic actions. Such a processor is composed by the following modules, that can be pipelined: a lexical analyzer (that tokenizes the input (source text)); a parser (that performs the syntactic analysis of the token stream to rebuild the derivation tree); a semantic analyzer (that evaluates the attributes according to semantic equations and checks for static consistency verifying contextual conditions); and the translator (that uses structural information and attribute values to produce the desired output).

The processor so far described takes two inputs—an attribute grammar and a source text (a sentence of the language defined by that AG)—and delivers as output some result derived from the source text.

That approach—*use of an AG to specify a language and its transformation, and a CG to produce the processor code*—saves programming time and is an error-pruning strategy; it is in fact a nice way to reuse optimized code.

We decided to apply this principle to our problem of processing file-system directories to catalog and transform documents. For that purpose we establish the following correspondence between grammar components and file-system components:

- files are terminal symbols;
- directories are non-terminal symbols
- the root directory is the axiom, or start-symbol;
- the structure of each directory (containing other directories and files) is described by productions, or derivation rules;
- all the information characterizing files and directories—i.e. type, time stamp, modes, etc.—is described by means of attributes associated with the

[3]Generally speaking, it is in fact a language processor generator

(terminal/non-terminal) symbols.

- semantic equations associated to the productions define how to propagate attribute values over the tree and how to compute the value of derived attributes;
- translation rules (based on attribute values, also associated with productions) specify the way the documents in the file-system directories will be transformed.

In a similar way, we can develop a tool to process file-system documents from the new formalism (introduced above) that we call DAG. Obviously that tool acts like a language processor generated from an AG.

The two last layers—the semantic analyzer and the translator—exhibit a behavior very close to their respective counterparts in a traditional language processor.

However, the lexical analyzer has no need to recognize terminals concatenating source text characters, because files are there with its name. Also the parser has a part of its task simplified, because the directory tree is already built; what happens is an operation like *tree-pattern matching with variable instantiation*. In some sense, in this case the lexical and syntactic analyzers resemble a structured editor (syntax-directed editor) [Saraiva, 2000, Reps and Teitelbaum, 1989, GrammaTech, 1995]: it knows the syntax tree for the input and just reads (get values) the terminals belonging to classes (those that are not simple syntactic sugar. but instead have semantic values).

To use the tool we need a language to describe the DAG. The meta-language for that is defined in the grammar bellow.

We chose a concise notation (Perl like) which means the language is very minimalist concerning syntactic sugar (no keywords) and also structurally minimalist (no declaration section). In other words, it follows a scripting language approach.

```
1  dag_file: rules

2  rules: rules attr_rules
3       | attr_rules

4  attr_rules: rule attrs

5  rule: IDENTIFIER ';'
6      | IDENTIFIER '--->' rhs ';'

7  rhs: rhs quantifier
8     | quantifier

9  quantifier: IDENTIFIER '*'
10           | IDENTIFIER '?'
11           | IDENTIFIER '+'
12           | IDENTIFIER

13 attrs:
14       | attrs attr

15 attr: '::' attribute
16     | '::' code

17 attribute: IDENTIFIER list ('='|'&') '{' PERL_CODE '}'
```

```
18 code: '{' PERL_CODE '}'

19 list:
20     | '[' dep_list ']'

21 dep_list: IDENTIFIER
22         | dep_list ',' IDENTIFIER
```

Reading the grammar, each file consists of a set of rules. These rules, like normal grammar rules, consists of a left hand side (the name of the non-terminal symbol — directory type) and a right hand side with the directory structure. This structure is nothing more than a sequence of symbols, each one with a quantifier. These identifiers represent each type of file or directory found. The quantifier represents the number of elements of that type we can found: any number of times (`*`), one or more times (`+`), none or one time (`?`) and if no quantifier is written, exactly one time.

For each rule we can write a section of code to be executed before any attribute calculation, and then any number of attributes equations. There are two special kind of attributes code: it can return the contents of the attribute or write it directly to disk. This difference can be noticed in the syntax, which can be '`=`' or '`&`'.

Each attribute can contain a list of dependencies. These specify the order attributes should be built.

3. Writing a grammar: an example

For a better understanding, let us see a simple example. Suppose that we want to collect photos taken in different places/countries around the world, and place them into separate directories, one directory for each continent and, inside it, a directory for each country.

Then, we will call `Continent` to directories which have directories inside, and `Country` to the other directories. We want to make, for each Continent, a HTML file with a list of countries inside. For each Country, it would be nice to make a HTML file with photo thumbnails and respective legends. In fact, we can include an XML document for each `Country` with a small legend for each photograph.

So, this structure contains at least four different types of files and directories: `Continent`, `Country`, `Photo` and `XML` (a file with photo legends).

Like in standard parser generation, we need a lexical analyzer and a syntax analyzer. On DAG, syntax is the directory structure accepted. The lexical analyzer is a function from file to type. This function will analyze each file name and return the file's type:

$$
\begin{array}{rcl}
typeof & : & filepath \longrightarrow type \\
typeof(x) & \triangleq & \begin{cases}
\text{if } isDir(x) \begin{cases} \text{if } hasDirs(x) & return(Continent) \\ \text{else} & return(Country) \end{cases} \\
\text{else} \begin{cases} \text{if } x \cong \text{'*.xml'} & return(XML) \\ \text{if } x \cong \text{'*.jpg'} & return(Photo) \end{cases}
\end{cases}
\end{array}
$$

Here is the grammar:

Attribute Name	File	Directory
foo	_foo_name	name/_foo_
.bar	__name.bar	name/__.bar
foo.bar	_foo_name.bar	name/_foo_.bar
!foo	_name/foo	name/_/foo

Table 1: Attribute creation for element name *name*

```
1 Continent ---> Country*;

2 Country ---> Photo* XML;
```

The files we want to generate can be viewed as attributes of the grammar. So, we can create attributes associated to the grammar symbols that are no more than simple HTML or image files, stored in the file-system.

As a implementation detail, attribute names are mapped to the real path in the file-system following a set of internal rules expressed in table 1. Each attribute has a file name beginning with "_" in order to be not confused with terminal symbols.

It is possible to associate attribute creation for each terminal or non terminal symbol from the grammar. This way, we can write a small perl function to create thumbnails. Using the Linux `convert` command, it is very easy.

When we are writing the attribute equations, we need to know the real path of the files to convert and the real path of the new files to be created (attributes). The Directory Attribute Grammars has a variable for the current rule information: `$_`.

```
1 Continent ---> Country*;

2 Country ---> Photo* XML;

3 Photo;

4 ::TN do{ ‘convert -geometry ’150x150>’ ’$_->{_}’ ’$_->{TN}’‘; }
```

This grammar differs from the previous one on the last two lines.
Line 3 – we are talking about the `Photo` type files.
Line 4 – defines the semantic equation for calculating the attribute `::TN`[4]. The four dots symbol (`::`) is just syntactic sugar.

In this case, the semantic equation has the form `do{...}` which means that the function creates the attribute file itself (in a imperative way).We will see another way semantic equations can work.

This semantic equation calls the convert command and the only strange thing is the variable `$_` that contains a reference for a hash. This variable represents the current rule object. The underscore element (`$_->{_}`) is the name of the file we are processing and the other (`$_->{TN}`) is the `::TN` attribute file name.

[4]We use TN to abbreviate thumbnail – a tiny view of the photo.

To create a thumbnail gallery for each country we can create another attribute but this time to the `Country` rule. If the directory name is *France* and we want to create an attribute inside it (as attributes start with an underscore) DAG will call it `France/_index_.html`.

```
Continent ---> Country*;

Country ---> Photo* XML;

::index.html ={
   join("\n",map { "<br><a href='".
                    get_legend($XML->{_},$_->{_}).
                   "'><img src='$_->{TN}'></a>" }
                  @Photo);
}

Photo;

::TN do{ 'convert -geometry '150x150>' '$_->{_}' '$_->{TN}''; }
```

The semantic equation for this attribute is a little more complex. The general form `={...}` means that the following function will return a string that will be the contents of the attribute file: the definition is done in a functional style. This way, we do not need to write code for opening, writing and closing the file.

The command is very simple. We are joining with a new line separator a set of strings; each of them refers to one `Photo` element inside the `Country` directory. Information of each file inside the current directory with type `Photo` is packaged on hashes witch references are allocated on the `@Photo` array.

We process each of the elements of the array with the `map` keyword that creates strings with HTML code for link creation and image inclusion. Note that on this string the `$_` variable is referring to each element of `@Photo` and not to the rule object. So, `$_->{_}` is the photo file name and `$_->{TN}` is the thumbnail file name.

Of course we can write some more HTML code to make a nicer output. Now, we want to make, for each continent, an index with a link to each country photo album. We will call this attribute `::index.html` although there is one with the same name for countries, but they are on different directories. The following grammar should do that!

```
Continent ---> Country*;

::index.html ={
   join("\n",map { "<br><a href='$_->{'index.html'}'>$_->{_}</a>" }
                  @Country
}

Country ---> Photo* XML;

::index.html ={
   join("\n",map { "<br><a href='".
                    get_legend($XML->{_},$_->{_}).
                   "'><img src='$_->{TN}'></a>" }
                  @Photo);
```

```
12  }
13  Photo;
14  ::TN do{ ‘convert -geometry ’150x150>’ ’$_->{_}’ ’$_->{TN}’‘; }
```

The grammar file to make HTML pages is now complete, and we can browse the photo gallery.

4. The directory attribute grammar processor

The DAG tool is implemented in Perl using PARSE::YAPP to make the syntactic analysis and using regular expressions for lexical analysis. It is a module that can be used within any Perl script.

Each attribute equation from DAG generates a Perl function. The semantic actions from DAG are true Perl code and work as functions. They receive a list of filenames and must return a file content (or, from another viewpoint, return a file). This list of filenames include the child file and attributes names, and the current rule filename and attributes already calculated.

The system works depth first, i.e., when processing a directory, it gets the directory contents and splits them into directories and file. It then calls itself recursively for every directory found. Attribute dependency between different generations (a father depending on a son) is therefore guaranteed and does not need to be checked.

Meanwhile, a method to solve dependency between brother attributes is under development.

There are some problems with error messages when the semantic equations have syntatic errors. We are developing another version, this time compiling (as `yacc`, for instance) to a perl source file that could be executed only if there are not errors.

Some other features not directly related to grammar exist. For example, attributes are evaluated only if they are not up to date, like standard Unix makefiles. On the other hand, if we would like to create this system using makefiles, we would need to create a makefile for each directory processing.

5. Measuring an Application

This formalism was used to build an Internet website for a virtual museum[Almeida et al., 2001], “Museu da Pessoa”. So, we used the file system to store and catalogue the documents to be presented. These include:

- interviews (more than 200 XML transcriptions);
- photos (over 500 images and XML legend files);
- project descriptions (XML files);

The grammar written (according to the meta-language formally introduced in section 2.) to set-up the museum from all this material only needs three production rules to define the basic directoriy types, as follows:

```
Project  ---> Project* Interview* Photo_album? Synopsis? HTML*;

Interview ---> Transcription* Photo_album? Sound? HTML* ;

Photo_album ---> GIF* JPG* Photo_legend?;
```

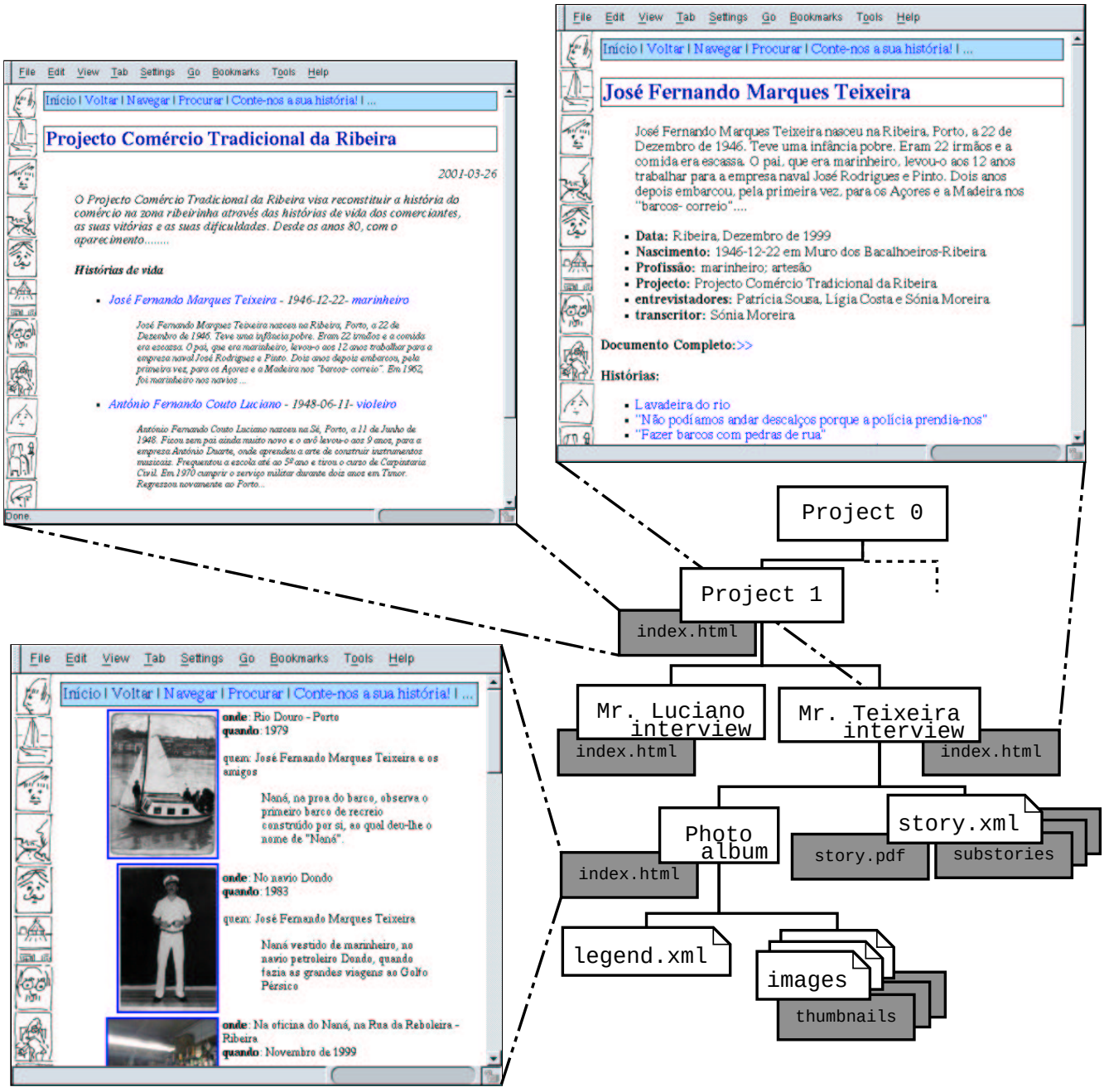

Figure 1: Museu da Pessoa's web museum

The terminal and non terminal symbols have more than 20 rules associated for attribute evaluation (to generate HTML pages for all kinds of documents, HTML index pages for photograph albums, catalogue files, thumbnails and, PDF and postscript files for each interview and book definition (for user download and read at home).

Figure 1 shows the file system structure for a project; it contains two interviews. One of them, contains a file (story) and another directory, with a photo album (images and legend) inside. The white blocks are directories, the white sheets are files and the shaded blocks are generated attributes. In the figure we can see a lot of attributes generated by the DAG processor from the grammar; the three pictures from HTML pages are attribute values, but are only some examples.

The following table shows some figures taken from this concrete project:

Tree size	283 Mbytes
Decorated tree size	534 Mbytes
XML files	267 files
DOC files	178 files
GIF and JPG files	$3 + 521 = 524$ files
HTML files	14 files
HTML attributes	1 745 files
JPG attributes	1 056 files
EPS attributes	533 files
PS attributes	223 files
Total attributes	4 926 files
Traverse the tree	2 minutes
Tree build from scratch	2:30 hours

The table does not show all the attribute categories. There are many attributes used only for catalogue building, conceptual navigation and some temporary attributes. The total number of files includes them as well.

We should make clear that the biggest slice of time used in the creation of the decorated tree from scratch is spent shrinking and converting images. The rebuild process after adding one more project (about 6 interviews and 10 photos for each of them) takes about 6 minutes.

It is impossible to present a quantitative comparison between this approach and a traditional one (hand building of the web-site) because we did not done such a thing; but we are sure that it would take much many hours. However, our claim is that this approach has all the benefits of an automated processor based on a formal specification (that represent all the work the user has to do).

6. Conclusions

DAG (Directory Attribute Grammars) formalism and processor was started as a simple exercise to simplify a complex real task in the context od XML documents publishing, but quickly it became an useful tool.

We think that DAG notation can be used to add formal level to file-system, in order do describe the file-system structure, and define how a set of new views should be built. We believe that DAG helps with the maintenance of archives supported by the file-system. We are porting other web sites to use this technology as dynamic pages served statically.

On the abstract, we said DAG = AG + Makefiles. Although we use the concept of makefiles to produce only necessary files, the use of makefiles instead of DAG is not a good idea. In fact, we would need to produce one makefile for each directory, which would be a painful operation. What we could really do is transform DAG in an automated makefile producer.

We did not present a comparison of processing times between DAG and other systems. The reason is that we do not know any other system like this, accomplishing such a task in a similar way.

Future work

A lot of work is planned to improve this tool, in order to make it more powerful, more useful, and more flexible. The future (and present) work include:

- improve the error handler: it should be possible to define various levels of errors. This way, we can have fatal errors (cause execution abortion) or just simple errors or warnings, reporting the problem to a log file, and resuming the exection; examples of this last type are files found whithout a type, or files found where that type is not allowed.
- build a tool – *flexer*(currently under development) – to generate `typeof` functions. This tool behaves like a lexical analyser generator of the tradicional compiler generator tools. This way, users should not need to write their own `typeof` function.

References

Alblas, H. and Melichar, B., editors (1991). *Attribute Grammars, Applications and Systems*. Czech Technical University – Prague, Springer-Verlag. Lecture Notes in Computer Science, nu. 545.

Almeida, J. J., Rocha, J. G., Henriques, P. R., Moreira, S., and Simões, A. (2001). Museu da pessoa – arquitectura. In *Encontro Nacional da Associação de Bilbiotecários, Arquivista e Documentalistas, ABAD'01, Porto*.

Christiansen, T. and Torkington, N. (1999). *Perl Cookbook*. O'Reilly and Associates, Inc.

Consortium, W. W. W., editor (10 February 1998). *eXtended Markup Language (XML) version 1.0 recommendation*. http://www.w3.org/TR/1998/REC-xml-19980210.html/.

Dencker, P. (1980). Benutzerbeschreibung des PGS. Interner Bericht 8/80, Institüt für Informatik, Univ. Karlsruhe.

Deransart, P. and Jourdan, M., editors (1990). *Attribute Grammars and their Applications*. INRIA, Springer-Verlag. Lecture Notes in Computer Science, nu. 461.

GrammaTech, I. (1995). *The Synthesizer Generator Reference Manual*, 4.th edition.

Gray, R., Heuring, V., Kram, S., Sloam, A., and Waite, W. (1990). Eli: A complete, flexible compiler construction system. Research report, Univ. of Colorado at Boulder.

Johnson, S. C. (1975). YACC yet another compiler compiler. Technical Report CSTR32, Bell Laboratories, Murray Hill.

Kastens, U. (1991). Attribute grammars in a compiler construction environment. In Alblas, H. and Melichar, B., editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 380–400. Springer-Verlag. LNCS 545.

Kastens, U., Hutt, B., and Zimmermann, E. (1982). GAG: A practical compiler generator. In *LNCS 141*. Springer-Verlag.

Kasters, U. (1994). Construction of application generators using eli. Research Report tr-ri-94-143, University of Paderborn.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145.

Lämmel, R. and Mernik, M. (2001). Special issue on domain-specific languages. *Journal of computing and information technology*, 9(4):i–iii.

Mernik, M., Lenic, M., Avdicausevic, E., and Zumer, V. (2000). Compiler/interpreter generator system lisa. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*.

Parr, T. J., Dietz, H. G., and Cohen, W. E. (1993). *Advanced Tutorial PCCTS 1.xx*. School of Electrical Engineering, Purdue University, version 1.xx edition.

Reps, T. and Teitelbaum, T. (1989). *The Synthesizer Generator Reference Manual*. Texts and Monographs in Computer Science. Springer-Verlag.

Saraiva, J. (2000). Language-based environments.

Sloam, A. M. (1994). Evaluation of automatically generated compilers. Research report, Department of Computer Science, James Cook University, Townsv ille.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages. Technical Report SEN-R0032, ISSN 1386-369X, CWI. an annotated bibligraphy.

Waite, W. and Carter, L. R. (1993). *An Introduction to Compiler Construction*. Harper Collins College Publishers.

Waite, W. and Goos, G. (1984). *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag.

Wall, L., Christiansen, T., and Schuartz, R. *Programming Perl*. O'Reilly and Associates, Inc.