Language Standardization Needs Grammarware

Steven Klusener, Vadim Zaytsev
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: {steven,vadim}@cs.vu.nl

October 4, 2005

Abstract

The ISO programming language standards are valuable documents that describe the syntax and semantics of mainstream languages. New features are proposed after thorough reviews by the standardization committees, leading to change documents that describe which modifications have to be enforced in the language standard document in order to actually add a new feature to the language. Maintaining these documents, both the language standard itself and all the change documents, is a time and resource consuming effort and in the evolution of these documents inconsistencies may be introduced. In this note we propose to utilize *grammarware*, a collection of new methods and new technology which can be used to support the advancement of these language documents in a more structured way. Besides, we will discuss how other tooling (like browsable language definitions, parser generators, pretty-printers, code checkers, etc.) can be obtained from the language standard. The final objective is threefold: (1) to facilitate the standardization committees in their activities and to raise the quality of the language standard documents; (2) to extend the usability of language standards by providing various presentations of each standard (in a human readable document, in a browsable form, in a machine readable BNF, etc.); (3) to help tool builders (compiler vendors, IDE vendors, etc.) in generating their parsing front-end, and to provide technology for tool builders to specify differences between their dialects and the actual standard.

1 Introduction

Software engineering is an established area, both in the IT industry and in the academic field of computer science. In the recent years a lot of progress within software engineering is made in (among others):

1. **Computer languages and specification formalisms,** with a continuous increase in the level of abstraction and declarativity;
2. **Tool support.** Software is not written in simple text editors anymore, but in advanced *Integrated Development Environments* (IDEs), like Microsoft Visual Studio, IBM WebSphere Studio and Eclipse, Borland Enterprise Studio, Anjuta DevStudio, etc.;

3. **The process of software development,** in order to be able to reuse as much as possible, to generate as many artifacts from existing data.

Defining a programming language (standard) is often considered as a "one time only" process, as most people are not aware of the dynamics of programming languages. However, programming languages (just like software) evolve over time. For certain languages this process is under control of the International Standardization Subcommittee for programming language, ISO/IEC JTC1/SC22 (see also `www.open-std.org/jtc1/sc22`). Such a language standard is a complex document that may consist of hundreds of pages (for example, the latest COBOL standard, ISO/IEC 1989:2002, is more than 800 pages long). Writing and maintaining such a document and keeping it consistent is as complex as writing and maintaining a large software system. Furthermore, tools like parsers and compilers are developed on a basis of a language specification and while developing these inconsistencies in the language standard may, or may not, be noticed and have to be resolved (which leads to inconsistencies, nonconformance and eventually language dialect development). The development of these tools is a very time consuming process, which have to be performed by every compiler vendor.

In this paper we take some lessons learned from software engineering, as mentioned above, and we combine them with existing *grammar engineering* results. We will not go into the details of a formalism for language specifications, although we do acknowledge the relevance of such a formalism. We will only briefly touch upon the BNF formalism and its extensions.[1] We will discuss the use of grammar engineering technology, also referred to as *grammarware*, such that the syntax definitions become consistent and such that various tools can be automatically generated from these syntax specifications. Furthermore, we will discuss an organized process in which consecutive versions are obtained by employing changes to the syntax definition of a language.

Objectives. The main purpose of the present paper is to put forth a large body of expertise and technology that is already available for some time. We will show that by formalizing the syntax definitions of programming languages and the processes by which these are extended and maintained, the quality of such syntax definitions can be improved and related tooling can be generated automatically.

This paper also discusses *separation of content and presentation*. In the current language standards the content of the standard is intertwined with its presentation. By separating these two aspects from each other one can maintain the content (the language definition itself) without being bothered by the presentation (e.g., the Adobe FrameMaker or `nroff` document). In our view the content of the syntax definition is maintained within some structure, from which various documents (in FrameMaker, `nroff`, XML, LaTeX, or some browsable form in HTML) can be generated.

This also allows for support of different *views* on the language standards. Engineers that build tools like parsers and compilers have a different view than software engineers that use the standard as language reference. Tool builders have to understand all the

[1] *BNF* is a very well-known format for describing grammars, standing for Backus-Naur Form or Backus Normal Form. John Backus introduced the format back in 1960s (as a part of his work on Algol 60), and Peter Naur contributed to it later.

details of the syntax (formally defined in the syntax rules) and maybe also have to know the semantics (informally described in text). Programmers that program in the language itself are less interested in the details of the syntax (as these details are implemented in an IDE developed by the formerly mentioned tool builders). These programmers are more interested in the rationale behind certain constructs; for example, if something can be implemented in various ways, what will be the most appropriate one?

Current state of affairs and future work. The paper will focus on the current state of affairs with respect to grammarware.At the end of this paper we will also briefly discuss whether and how this approach can be further adopted within the ISO community.

Syntax definition, not semantics. This note will focus on the syntax definitions of programming languages, it will not address the definition of the semantics of programming languages. Formalizing the semantics is outside the scope of this paper, it does require different and considerably more complex techniques.

2 Grammar engineering in Amsterdam

In Amsterdam there is a long tradition in research in the area of language construction and language based tool development. It involves the University of Amsterdam (UvA), the Center of Mathematics and Computer Science (CWI) and the Free University of Amsterdam (VU). The research groups do work closely together in the Software Engineering Amsterdam (SEA) initiative, which is partially founded by the Dutch scientific organization (NWO). Currently the research group at the UvA focuses its effort on semantics of programming languages, the research group at the CWI works on *generic language technology* (as will be explained below), and the research group at the VU deals with grammarware engineering, reverse engineering and IT portfolio management. These related topics provide a solid basis for study of languages, including definition of their syntax.

Grammarware. In [12] our colleagues introduce the term *grammarware*, comprising grammars and all related grammar-dependent software: a parser, a compiler, a pretty-printer, a lexical analyzer, a run-time environment, a development environment, a browser, a type checker, a structural editor, a loader, a debugger, a preprocessor, a profiler, as well as an interpreter and other numerous reverse engineering tools, code refactoring tools, slicing tools, (re)documentation tools, software analysis tools, static checkers, optimizers, etc. They argue that grammars are everywhere—for example, XML Schema schemas and DTDs that describe the structure of XML data are in fact grammars, which makes all XML based tools (parsers, validators, data binding tools, XSLT, etc.) grammarware. In their paper they present a research agenda regarding grammarware, while in this note we discuss the grammarware issues more specifically in the context of language standardization.

Grammar Recovery and Browsable Grammars. A grammar forms the very basis of a language definition. From a grammar, tools like parsers can be generated automatically. However, there is a large difference between a human readable language reference (either an ISO Language Standard or some compiler vendor specific Language Reference Manual) and a formal grammar. While language references are widely available on the Internet, formal grammars are not. Ralf Lämmel (formerly VU/CWI, now at Microsoft Research) and Chris Verhoef (VU) have shown in their paper [14] how an IBM VS II COBOL grammar can be obtained from the language reference manual made by IBM. The result of that research is made available via [1], it is the first (and still the only) COBOL grammar available on the web. This can be checked easily by googling for `Cobol grammar`.

Grammar Adaptation. When a grammar is recovered from a language definition, inconsistencies often have to be removed and stylistic improvements have to be enforced. In his paper on grammar adaptation [13] Ralf Lämmel introduces a collection of operations which can be applied to a BNF grammar, resulting in an adapted BNF grammar.

These grammar operations can also be used to specify differences between a compiler specific language dialect and a language standard. In our research group we use this technique, for example, to obtain COBOL dialects' grammars from our COBOL base line grammar.

Grammar Definitions and Parsing Technology. Grammars define the formal structure of a language, hence they define as well the structure of a piece of source code. The structure of the source code is typically represented by a so called *parse tree*. A *parser* is an algorithm that takes a piece of source code as an input and constructs the parse tree. If the source code does not conform to the grammar, a parser reports some *parse error*.

Parsers can be generated from grammars, by so called *parser generators*. For an overview of the different parser generators (top-down parsing, LL, LR, LALR, SLR, GLR, etc.) we refer to standard text books like [2] or [5]. One of the most popular parser generators is called YACC [10], Yet Another Compiler Compiler, which was first introduced in 1975. The point is that YACC, just like many other parser generators, requires that the grammar satisfies specific conditions. (In technical terms, the grammar must be free of *conflicts*. It must be clear which alternative has to be expanded (up to a certain lookahead), and it must be clear when an alternative is finished.) In other words, YACC does not support all BNF grammars, and in order to use YACC one has to manipulate the grammar until it satisfies the conditions (i.e., until all conflicts are removed). This, however, can decrease the readability of the grammar considerably. Manipulating the grammar such that it fits within the boundaries of a specific parser generator is therefore also known as *grammar hacking*.

One of the first grammars around was introduced by Kernighan and Ritchie in their book about C programming language [11]. Their grammar is a typical YACC grammar, it falls within the limits of YACC. In [18] it is motivated that the declarativity and readability of such grammars have been sacrificed considerably. One other limitation

of that grammar is that iterative constructs (such as a list of statements) have to be defined recursively, instead of using the iterative constructs that became standard in *Extended* BNF. This C grammar, however, has set the scene, many grammars since then have followed its style. One recent example is the C# language standard [9] which, for no other than historical reasons, stays within the YACC limits although the YACC dependency is nowhere mentioned (and YACC claimed not to be used by the grammar creators).

Grammars are usually considered extremely static objects, made once and not subject to any change. This is apparently not completely true, especially in the context of the ISO standardization committees that continuously improve their programming languages. Also for reverse engineering purposes one need to adapt all the tools (a grammar and a parser in particular) to be able to work with a different dialect. Changing and maintaining a grammar is extremely expensive (in terms of man-effort), in many tool building organizations the required knowledge is limited to a small number of specialists. Some reasons for this situation are: (1) the grammars are not declarative but already geared towards the YACC-like style (or some other limited grammar format), (2) the tools are implemented on a low-level. For a more in-depth discussion about this we refer to [18], the title *Current parsing techniques considered harmful* says enough. Its message is that nowadays one should use a parser generator that supports all context free grammars, i.e., a parser generator that puts no limitations on the grammar.

In our research group we use scannerless generalized LR (SGLR) parser engine of the ASF+SDF Meta-Environment from the CWI. The parsing technique supports all BNF grammars (no grammar hacking is required).

EBNF as a Formalism for Defining Syntax. For many decades BNF has been the standard for defining the syntax of programming languages. First it was introduced to define the syntax of Algol, later it has been used to define the syntax of many other languages as well. The original version of BNF is rather restricted (for example, every repetition of items had to be expressed by recursion), this has been resolved by adding certain operators for repetition and for optional syntax constructs. This extension is known as EBNF (Extended BNF).

EBNF can be extended further to express syntax constructs in a more declarative way, for example when a list of items has a separating symbol (like a comma or a semicolon). Other extensions one can think of is a notion of modularity or the use of disambiguation constructs. The syntax formalism SDF (a part of the above mentioned ASF+SDF Meta-Environment) can be considered as such an extension of EBNF. In this note we will not address these issues anymore, we simply assume that we have some sort of extended version of BNF.[2]

Generating Tools from Language Definitions. We have already mentioned that parsers can be generated from grammars. Many other tools can be generated from grammars, directly or indirectly, such as:

[2]Note that the COBOL language standard does not use BNF. It uses syntax diagrams, which can be considered a very extended and very liberal form of extended BNF.

- Tools that check that certain constructs, like *obsolete* keywords, are not used;
- Tools that migrate source code from one language variant to another, as in [15] where a collection of rules was introduced to go from COBOL'74 to COBOL'85 and [20] where these rules were extended and integrated into an algorithm that can be used on a large scale (millions of lines of code);
- Tools that enforce certain layout standards, which are called *pretty-printers* [17, 19];
- Browsable versions of a language definition, as described above.

Within the ASF+SDF Meta-Environment tools such as listed above can be specified with the use of so called *rewrite rules*, given which the Meta-Environment generates tools automatically.

3 Other related work

Unlike the previous section where we enlisted research activities of our group and related groups in Amsterdam, this section is going to give brief overview of the work of others on adjacent subjects.

The ISO standard for Extended BNF. The ISO/IEC Standard 14977:1996(E) [7] defines a standard for the grammar notation EBNF (Extended BNF). It formalizes the conventions that are normally used for EBNF. We particularly quote and acknowledge the following objectives of that standard:

- *"It is desirable that a standard syntactic metalanguage should be (...) concise, so that languages can be defined briefly and thus be more easily understood."*

  The current standards for COBOL and C#, for example, use different mechanisms, namely syntax diagrams and BNF respectively.

- *"Standard syntactic metalanguage should be (...) formal, so that the rules can be parsed, or otherwise processed, by a computer when required."*

  We consider this an extremely important requirement, which currently does not yet hold for the COBOL standard [8], for example.

- *"Standard syntactic metalanguage should be (...) precise, so that the rules are unambiguous."*

  Although being precise is not sufficient, also with a precise grammar formalism one can easily specify ambiguous grammars as we will show below.

Structured data and XML. The eXtensible Markup Language also known as XML [4] is the world-wide standard for structured data. In an XML file the structure of the data is determined by *tags*, hence, an XML file corresponds to a textual representation of a *parse tree*. XML files belong to document types, which can be defined by means of DTDs or XML Schema schemas [3, 16]. Schemas and DTDs define the structure of an XML file, which means they are in fact grammars. If an XML file is properly constructed and it conforms to the associated grammar, then it is said to be *valid against the schema or DTD*. This check is done by means of an *XML validator*. Note however that it only verifies the structural validity of the XML file (which, as mentioned before, corresponds to a parse tree). Performing such a check is much simpler than constructing a parse tree from a piece of text.

So, most of the XML concepts are covered somehow by grammarware concepts (i.e., grammar and parsing technology). Still there is the question why XML has taken over the world in a few years without much notice of the state of grammarware? Moreover, *structured data* may be technically covered by *structured texts* (described by grammars, like computer programs), but still we have to understand when XML is the appropriate technique and when do we have to use grammars.

For example, an assignment like `x := x + 1` can be expressed in XML (given some DTD) as

```
<statement>
 <assignment>
  <lhs>
    <id> x </id>
  </lhs>
  <rhs>
   <expr>
    <plus-expr>
     <left-op>
       <exp><id>x</id></exp>
     </left-op>
     <rigth-op>
       <exp><int>1</int></exp>
     </rigth-op>
    </plus-expr>
   </expr>
  </rhs>
 </assignment>
</statement>
```

It is unthinkable to construct programs like this. The XML representation takes 19 lines for our assignment, and this is only a simple statement. Imagine how a more complex statement like

```
obj1.attr := obj2.method(x+1,bool1 or bool2)
```

is represented in XML.

On the other hand, it is a bit over the top to introduce a grammar for dates:

```
date  : year "-" month "-" day ;
year  : int-4;
month : int-2;
day   : int-2;
```

and to be forced to use one of the many parser generators to be able to parse a simple date like `2005-09-07`. In this case it makes more sense to write the date directly in XML as

```
<date>
 <year>2005</year>
 <month>10</month>
 <day>07</day>
</date>
```

Generally, XML works good for semi-structured texts or texts that all have one consistent structure that does not change. That is why we do not want to use XML for programs, but due to the same reason we use it for interface descriptions, configuration files, etc (for some things like date formats or numbers the overhead of grammars and parsing is too large). Within ISO, programming languages are typically described with grammars (in BNF-like formalisms) while XML is used in other areas. It may be possible to define a language by a schema, but the result will not be found convenient by most programmers as we have shown earlier. Between XML and grammars there is some grey area (which still has to be investigated) where the choice is not so obvious and these two techniques compete. For this grey area it seems appropriate to study whether XML Schema schemas can be derived from (E)BNF grammars. This requires more research on the question whether this mapping puts any condition on the expressivity of the BNF dialect that is used.

```
program
 : "program" id
       declarationSection
   "begin"
     statements
   "end" id?
 ;
```

```
declaration-section
 : "declare" decl ","* ";"
 ;

decl
 : id ":" type
 ;
```

```
statement
 : id ":=" expr
 | id ":" "{" statements "}"
 | "goto" id
 | "if" expr
      "then" statements
      "else" statements
   "end-if"
 | "read" id
 | "write" id
 ;
```

```
type
 : "int"
 | "string"
 | "bool"
 ;

primary-expr
 : id
 | integer
 | string
 | "(" expr ")"
 ;
```

```
and-expr
 : primary-expr
 | and-expr "&&" primary-expr
 ;

or-expr
 : and-expr
 | or-expr "||" and-expr
 ;

multiply-expr
 : or-expr
 | multiply-expr "*" or-expr
 ;
```

```
divide-expr
 : multiply-expr
 | divide-expr "/" multiply-expr
 ;

plus-expr
 : divide-expr
 | plus-expr "+" divide-expr
 ;

expr
 : plus-expr
 | expr "-" plus-expr
 ;
```

Figure 1: Syntax definition from language definition of PROTO, version 1.0

4 Current state of affairs

The current state of affairs of grammarware with respect to language standardization is explained with a running example, the toy language PROTO. In this running example we discuss the various cases that one typically finds in language definitions.

We will show how the language definition is corrected by a sequence of steps that can be applied automatically to the original language definition. We will also discuss how a next version of the language standard can be obtained from a current one en we will discuss how obsolete language constructs can be removed from actual source code.

A source code example. To give the reader a flavor of our toy language we first give a brief source code example:

```
program SIMPLE
 declare x : int, y : int ;
 begin
  body : {
    read x ;
    y := x + 1
  }
 end
```

The language standard of PROTO in BNF. The syntax of our toy language is taken from its fictitious language definition, it is given in Figure 1. In this figure only the syntax definition is given in BNF productions. (Note that the COBOL language definition doesn't use BNF productions but syntax diagrams, which correspond to a very liberal and a very extended version of BNF.)

In a real language definition, each syntax rule is explained with one or more pages of text. These texts are left out here.

Applied corrections. Below we discuss a sequence of improvements that are typically applied to the grammars in our language definitions.

1. **Fix misprints.** There is a mistake in "declarationSection", which has to be repaired:

   ```
   %rename declarationSection %into declaration-section
   ```

   These `%rename` commands can be applied directly to the BNF grammar definition, which will give us the corrected BNF. For real grammars like C of COBOL we may have to fix many of these misprints.

2. **Define missing non-terminals.** For certain non-terminals no syntax rules are given in the language definition, these missing definitions can be detected easily. We construct a directed graph, each non-terminal is a node and if a non-terminal `B` is used in the syntax rules of a non-terminal `A`, then we have an arrow between `A` and `B`. For example, the node `program` has three outgoing edges, one to `id`, one to `declarationSection` and one to `statements`.

   If all nodes are *reachable* from the top node, in our case `program`, then we know that there are no missing syntax rules. However, in our case we will see that that there is no edge between `statements` and `statement`. In fact, `statement` appears not to be used in any syntax rule. Hence, it has no ingoing edges and as such it is an unexpected top node, from which we conclude that the definition for `statements` is missing. In general, by detecting all these unexpected top nodes we can get the missing definitions.

How do we come to the missing definition of our non-terminal `statements`? Well, we take the (fictitious) language definition and we read in the text that *statements are separated by a semicolon ";"*. In our BNF variant, following [6], we can express this as follows:

```
%resolve statements : {statement ";"}* ;
```

The standard notation `S*` (meaning zero or more `S`'s) is extended here with the separating symbol ";".

3. **Remove YACC style.** When the grammar is properly defined (all non-terminals are defined and reachable from the top node), we can often make some more stylistic improvements. One very common improvement is called *de-YACC-ification*.

   As explained before, YACC is the standard parser generator that is known for many years. This parser generator requires that the grammar satisfies certain properties, one of them is that the priority of operators is expressed in an hierarchy of non-terminals. Thus BNF productions like

   ```
   expr :
     ...
     | expr "*" expr
     | expr "/" expr
     ...
   ```

   are not allowed.

   In Figure 1 we see a typical YACC style in the definition of the expressions. The non-terminal `expr` is defined in terms of `plus-expr`, which is defined in terms of `divide-expr`, etcetera. In other words, the priority between the operators (logical and, `&&`, binds the strongest, the minus, `-`, binds the weakest) is defined by a hierarchy of 5 auxiliary non-terminals.

   As we are free of YACC specialization, there is no reason at all to adopt this YACC style within our grammar. (Note, however, that this YACC style is very persistent as also the recent C# grammar uses it!). Therefore, we remove it by applying the `%redefine` command from Figure 2. Note that the underlying hierarchy from `plus-expression` to `unary-expression` becomes disconnected.

   The priorities between the various operators are defined elsewhere in our grammar, we will not discuss that here.

This refactoring of the grammar is also reflected in the parse trees, as given in Figure 3, where the parse tree of the arithmetic expression `x + 1` is given. That final one (on the right hand side) looks more natural, the parse tree which is obtained before the de-YACC-ification (on the left hand side) looks rather degenerated.

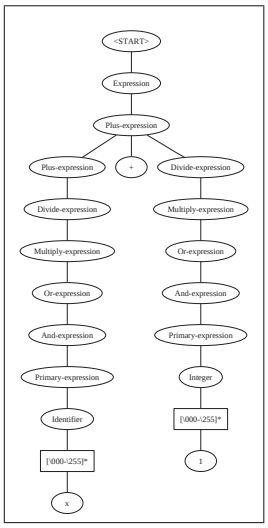
The following table summarizes the several steps in the correcting and refactoring process and the intermediate and final versions that come with it.

```
%redefine
expr
 : expr
 | plus-expr "-" plus-expr
 ;
%to
expr
 : identifier
 | integer
 | string
 | "(" expr ")"
 | expr "&&" expr
 | expr "||" expr
 | expr "*" expr
 | expr "/" expr
 | expr "+" expr
 | expr "-" expr
 ;
```

Figure 2: Removing YACC-style in favor of declarative (recursive) expression syntax

Some real-life cases. In Table 2 we give a brief overview of the grammars that have been corrected and refactored within our research group. (We have not included PL/I here, although it is present at [1], because it is not refactored with our current grammarware infrastructure.)

Note that even recent languages like C# still require a great deal of refactoring.



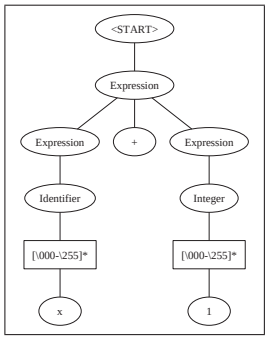


Figure 3: Parse tree, left after correction only, right after correction and refactoring

Version	Parse Tree	Steps
Language standard	Not available	Initial version
Corrected version	see 3, left	Mistypings corrected, missing productions added
Refactored version	see 3, right	De-YACC-ification, stylistic improvements

Table 1: Three versions of our language definition

In the table below LOC means number of lines of code of grammar commands.

Language	LOC	Reference	URL
C	652	ISO/IEC 9899	`www.nirvani.net`
COBOL	3380	IBM VSII	`publibz.boulder.ibm.com`
C#	1534	ISO/IEC 23270-2003	`isotc.iso.org`
Fortran	543	Cray Volume 3	`telematica.cicese.mx`
PROTO	46	This document	

Table 2: An overview of grammar refactorings

A new language standard, PROTO 2.0. Let us assume that our toy language PROTO has an active working group, WG99. This working group proposes to remove the label construct and the `goto` statement from the language. The `while-do` statement is added instead, as is given in Figure 4, which is taken from a fictitious change document.

Migrating the source code. One of the benefits of grammarware, is that we not only can describe how one language version should evolve into another one, but that we can define as well how obsolete source has to be changed into the new language version. In Figure 5 we see an example of a pattern that can be applied to PROTO 1.0 source code, when it matches each `goto`-fragment is rewritten into an associated `while-do` fragment.

Intermediate conclusions.

- With the current state of the art we can already define the structure of a language in a formal way, readable by humans *and* tools.

```
%exclude
statement
 : id ":" "{" statements "}"
 | "goto" id
 ;
```

```
%include
statement
 : "while" expr
     "do"
       statements
     "end-do"
 ;
```

Figure 4: Left: removing the obsolete `goto` statement. Right: the adding corresponding `while-do` statement.

Obsolete PROTO 1.0 pattern	Corresponding PROTO 2.0 pattern

`id :`
`  statements-1`
`  if expression`
`  statements-2`
`  goto id`
`  else`
`  statements-3`
`  end-if`
`  statements-4`
	`id :`
	`  statements-1`
	`  while expression do`
	`  statements-2`
	`  statements-1`
	`  end-do`
	`  statements-3`
	`  statements-4`

Figure 5: Syntax pattern that changes obsolete source code fragments into PROTO 2.0 compliant code

- We need Grammar Engineering:
  - Such that we can enforce changes to grammars in a controlled and repeatable manner;
  - Such that we can define a collection of checks that improve the quality of the grammar;
  - Consisting of a collection of grammar writing guide lines (such as: DO NOT YACC!)
- We need Generic Language Technology:
  - For parsing, i.e. generating parse trees out of source code (plain text);
  - Pretty-printing, i.e., formatting the layout and indentation;
  - Code analysis, such as type checking;
  - Code transformations, for the removal of obsolete language elements;
  - To support language prototyping when developing new language constructs.

5 Future work

- Extend Extended BNF further. We need a practically aimed, technology independent, modern and powerful formalism for describing everything needed for a modern language definition. The extension would perhaps go beyond "syntactic sugar", they can bear something like static semantics or other mechanisms for the goals we set in this note.
- Design a formal structure for language definitions. Costs (mostly human effort and time costs) can be shrunk by introducing one general format for all ISO programming language standard documents. The tools to support this format and migration from the existing infrastructure should be built too.
- Integrate current technology with other grammar engineering activities and generic language technology such that complete life cycle of a programming language can covered.
- Combine this generalized programming language life cycle technology with existing programming and development environments, integration into various third parties' products (mentioned earlier in this paper).

6 Grammarware adoption plan for ISO

In the previous sections we have explained what grammarware is and how it can be applied to language standardization and how the quality of the language standards can be improved while decreasing the total amount of required human effort. Not only the time effort in the language standardization can be reduced but also the time effort that is needed at the side of the compiler vendors

It is our objective to start a project in which prototypes are developed and evaluated by (members of) the ISO/SC22 community. If we continue this exercise with a toy language like PROTO, there is the risk that also our project will become a toy project for which there is no real interest. However, if we start with a large language definition like COBOL, then there is the risk that every step will take too much effort and too much time. Before we jointly come to the appropriate case study, we give the following possible steps:

1. Select one language standard as a case study, provide the language standard document in some textual format: plain text, HTML, XML, MIF, etc.[3]
2. Apply the corrections as discussed in the previous section of this document. Note that some of them regard real, although minor, errors in the language standard, like misprints, whereas other corrections have a more stylistic nature.
3. Generate browsable versions of the language standard, not only taking the syntax rules into account, but also the complete text and its structure.

[3] *MIF* stands for Maker Interchange Format, it is an ASCII based alternative representation for Adobe FrameMaker.

4. Generate a parser, such that source code examples can be parsed (and hence, can be checked).

5. Define other tools like *pretty-printers*, *code checkers* or *code improvers*.

6. Formalize some change documents, and apply them to the language version "new style".

Before such a project can start, some more issues have to be addressed.

1. Who will participate in the project?

   Applying the steps described above to a certain grammar is not real scientific research anymore. We know how to do it, and some of the underlying results have already been published (for example, see [14]). As our research group has to restrict itself to only research activities that can lead to new scientific publications, it is acceptable for us to do the same thing over and over again. However, we will be pleased to help others to use our results.

2. Is there funding for these activities?

   Apart from the first question we have to ask who would be willing to sponsor such a project. Are there any companies interested in the results? Do we strive for some open source schema, if so, are there any funding programs for that?

3. Are there other technical details?

   We cannot make prototypes which run on all existing platforms. Do we opt for Linux/Unix environment, for Windows operating system, for .NET, of some other platforms as well? Who will evaluate the prototypes? Who is going to test them?

4. Which results can be published freely, and which have to be distributed via the regular ISO channels? If we generate a browsable version of a language standard, can it be published on the world wide web, or can it only be distributed within the ISO working groups?

References

[1] VU Browsable Grammars Website. `http://www.cs.vu.nl/grammars/browsable`.

[2] A. V. Aho, R. Sethi, and F. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[3] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. *W3C Recommendation*, 28 October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028`.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). *W3C Recommendation*, 04 February 2004. `http://www.w3.org/TR/2004/REC-xml-20040204`.

[5] D. Grune and C. Jacobs. *Parsing Techniques*. Ellis Horwood, England, 1990. The second edition is upcoming, see also `http://www.cs.vu.nl/~dick/PTAPG.html`.

[6] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[7] ISO/IEC 14977:1996(E). Information Technology—Extended BNF. Available via `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`.

[8] ISO/IEC 1989:2002. Information Technology—Programming Languages—COBOL, 2002.

[9] ISO/IEC 23270:2003. Information Technology—C# Language Specification, 2003.

[10] S. C. Johnson. *YACC—Yet Another Compiler Compiler*. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Bell Laboratories, Murray Hill, New Jersey, 1978.

[12] P. Klint, R. Lämmel, and C. Verhoef. Towards an Engineering Discipline for Grammarware. *ACM TOSEM*, May 2005. To appear; on-line since July 2003, 47 pages.

[13] R. Lämmel. Grammar Adaptation. In *Procedings Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[14] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[15] M. P. A. Sellink, H. M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS Legacy Systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 72–82. IEEE Computer Society Press, 1999. Available at `http://www.cs.vu.nl/~x/cics/cics.html`.

[16] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition. *W3C Recommendation*, 28 October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028`.

[17] M. G. J. van den Brand, A. T. Kooiker, N. P. Veerman, and J. J. Vinju. An Architecture for Context-sensitive Formatting. In *International Conference on Software Maintenance*, 2005.

[18] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *International Workshop on Program Comprehension*, 1998.

[19] M. G. J. van den Brand and E. Visser. Generation of Formatters for Context-free Languages. *ACM Transactions on Software Engineering Methodology*, 5(1):1–41, 1996.

[20] N. P. Veerman. Revitalizing Modifiability of Legacy Assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.