BoB: Best of Both in Compiler Construction – Bottom-up Parsing with Top-down Semantic Evaluation

Heinz Dobler
Fachhochschule Oberösterreich
27 PUBLICATIONS **76** CITATIONS
SEE PROFILE

Wolfgang Dichler
1 PUBLICATION **0** CITATIONS
SEE PROFILE

BoB: Best of Both in Compiler Construction – Bottom-up Parsing with Top-down Semantic Evaluation

Wolfgang Dichler[1] and Heinz Dobler[2]

[1] **Software Engineering Department, University of Applied Sciences Upper Austria, Hagenberg, Austria**
w.dichler@gmail.com

[2] **Software Engineering Department, University of Applied Sciences Upper Austria, Hagenberg, Austria**
heinz.dobler@fh-hagenberg.at

Abstract

Compilers typically use either a top-down or a bottom-up strategy for parsing *as well as* semantic evaluation. Both strategies have advantages and disadvantages: bottom-up parsing supports LR(k) grammars but is limited to S- or LR-attribution while top-down parsing is restricted to LL(k) grammars but supports L-attribution. The goal of the work described herein was to combine the advantages of both strategies. The result is the compiler generator *BoB*, mainly a preprocessor for Flex and Bison (modern versions of Lex and Yacc). BoB processes compiler descriptions written in *Cocol4BoB* which supports L-attributed LALR(1) grammars and generates input files for Flex and Bison. Compilers generated by the BoB-Flex&Bison toolchain use bottom-up parsing and top-down semantic evaluation. So developers do not have to struggle with LL(1) conflicts and can use inherited as well as synthesized attributes in semantic actions. Another benefit of BoB is its simple yet powerful compiler description language.

Keywords: *Bottom-Up Parsing, Top-Down Semantic Evaluation, Attributed Grammars, Compilers and Compiler Generators.*

1. Introduction

The first compilers (e.g., for Fortran [2]) were “handcrafted” without extensive formal specifications and the usage of special tools. Later, compilers were created from attributed grammars (ATGs) [12] for defining the syntax as well as the semantics of the source languages, still writing parsers by hand using recursive descent [1] as a simple but efficient method for parsing. The developments in the last decades improved this error-prone manual transformation, as tools have been developed which support the compiler writers by generating (main parts of) their compilers from ATGs. These tools are called *compiler compilers* or *compiler generators*. These terms are misleading because they imply that full compilers can be created, but usually these tools create only some important parts of the frontend of a compiler, typically scanners and/or parsers (for syntax analysis and evaluation of semantics).

Over the years, mainly two categories of compiler generators have been created [19]:

1. Tools that generate compilers which use a *top-down* strategy (e.g., recursive descent) for parsing and semantic evaluation, where parsing is restricted to LL(k) grammars but semantic evaluation supports inherited, transitional and derived attributes (so called L-attributed grammars).
2. Tools that generate compilers which use a *bottom-up* strategy for parsing and semantic evaluation, where parsing supports LR(k) grammars but semantic evaluation cannot deal with inherited and transitional attributes (so called S- or LR-attributed grammars).

As mentioned above, both strategies have advantages and disadvantages. Therefore, compiler developers typically have to enter into compromise.

In 1986, Mössenböck described in [14] a new compiler generator called *Smart* which offered an interesting alternative to this situation which overcomes the limitations described above: compilers generated by Smart use a top-down parser and evaluate semantics in a bottom-up manner. But Smart was written in and for Modula-2 [23] and today this language is buried in oblivion (e.g., see [21]), thus Smart has not been used on a wide scale.

2. Problem Definition and Aim

Apart from the complicated syntax of their input languages for describing compilers in form of ATGs, current compiler generators still have significant disadvantages: either they do not support LR(k) grammars or they do not support inherited and transitional attributes in semantic actions (as used in L-attributed grammars). Therefore, compiler writers still have to choose between bottom-up and top-down parsers with corresponding semantic evaluators and have to deal with their drawbacks.

The aim for solving this problem, was not to develop a completely new compiler generator (as Mössenböck did with Smart) but to clever reuse existing compiler generators, an idea Katwik already presented in [11] as “a poor man’s approach for parsing attributed grammars”. In more detail, to aim was to create special kind of preprocessor called *BoB* (an acronym for Best of Both strategies) for Flex [16] and Bison [8] which are modern versions of the well-known and widely used tools Lex [13] and Yacc [10]. The resulting toolchain, using BoB prior to Flex and Bison (BoB-Flex&Bison) should be able to generate form L-attributed LR(k) grammars (supporting inherited, transitional and derived attributes for semantic actions) the main parts of the frontend for compilers in C++ [20]. So the overall aim of this work was to combine the best strategies for parsing and semantic analysis: the powerful bottom-up parsing with the powerful top-down semantic evaluation.

Another goal was related to the input language: BoB should use syntax for its input language (necessary for the definition of ATGs to describe compilers) which is deliberately not derived from the cryptic input languages for Flex and Bison but inspired by the simple and readable syntax of *Cocol-2*, the input language for *Coco-2* [6]. Therefore, Bob’s input language is called *Coco language for BoB* or *Cocol4BoB* for short. Cocol4BoB should allow the definition of LALR(1) grammars with semantic actions in C++ that obey the restrictions of L-attribution so that BoB can translate theses ATGs into valid input files for Flex and Bison, which are then used to generate the final source files (in C++) for the main compilers parts: scanner, parser and semantic evaluator. The resulting compilers use deterministic finite automata for scanning, bottom-up parsing and top-down semantic evaluation. For the implementation of the semantic evaluator the idea of Mössenböck incorporated in Smart have been taken up again.

3. State of the Art

This chapter gives a short overview of only those tools and the concepts which are necessary for the design and the implementation of BoB.

3.1 Compiler Generators

Bison [8] is a modern version of the well-known and widely used parser generator *Yacc* [10]. Bison from S-attributed LALR(1) grammars generates bottom-up parsers. Bison-generated parsers need scanners for reading the input stream and tokenizing the characters. *Flex* is a modern version of the well-known and widely used scanner generator *Lex* [13]. Flex generates from regular expressions scanners which are based on deterministic finite automata (DFA). The combination of (F)Lex and Bison/Yacc is widely used. All these tools are in and for C or C++.

Coco-2 [6] and *Coco/R* [15] are compiler generators based on the predecessors *Alex* and *Coco* [17]. Both tools accept as input L-attributed LL(1) grammars and mainly generate top-down parsers. The most important difference between both tools lies in the top-down parsing technology:

- Parsers generated by Coco-2 use an interpreter for *grammar code* (*G-code*, a special kind of byte code defined for Coco and extended for Coco-2) which supports flexible error recovery [7].
- Parsers generated by Coco/R use the classic recursive descent method which makes them highly efficient.

Besides parsers, Coco-2 and Coco/R can generate scanners based on DFA. Both tools are available in and for several programming languages (including C and C++).

The compiler generator *Smart* [14] uses bottom-up parsing and top-down semantic evaluation in the generated parsers. Smart is written in and for Modula-2 [23]. Nowadays, this programming language has no large spread [21] any more, thus Smart has not been used on a wide scale.

3.2 Compiler Description Languages

Compiler generators typically define their own compiler description languages. Usually, some kind of regular expressions is used to define the lexical structure (set of tokens) of the input language for the compilers to be generated and some kind of *Backus-Naur Form* (BNF) [3] is used for the definition of the syntax of the input language. Examples are the compiler description languages for the scanner and parser generators Flex and Bison.

Coco-2 [6] takes a different approach: its compiler description language *Cocol-2* (*Coco language*) is used for the description of both parts (scanner and parser), so *Extended Backus-Naur Form* (EBNF) [9, 22] can be used for the definition of the lexical structure (set of tokens) as well as for the syntax of the source languages of the compilers to be generated. This leads to consistent, shorter and more readable descriptions within a single document, avoiding duplicate descriptions for the scanner and parser in different notations and decreasing the probability of errors.

4. Design

This chapter gives an overview of the solution approach taken for BoB, lists the requirements for and presents a sketch of its input language Cocol4Bob and finally explains the Bob’s architecture as well as the architecture of compilers generated by BoB.

4.1 Solution Approach

Figure 1 presents a comparison of the power of the LL(k) versus the LR(k) syntax analysis strategies. The question visualized in Figure 1 is, which sequence α has to be chosen for the nonterminal *A*. Comparing the information available for answering this question ("known: ….") – in the left side for top-down analysis and in the right side for bottom-up analysis – makes it evident, that top-down analysis has less information during processing the stream of tokens as bottom-up analysis, which additionally has all the tokens of α which already have been pushed onto the stack of the PDA. Simply put: more information allows better decisions, so LR(k) is more powerful than LL(k).

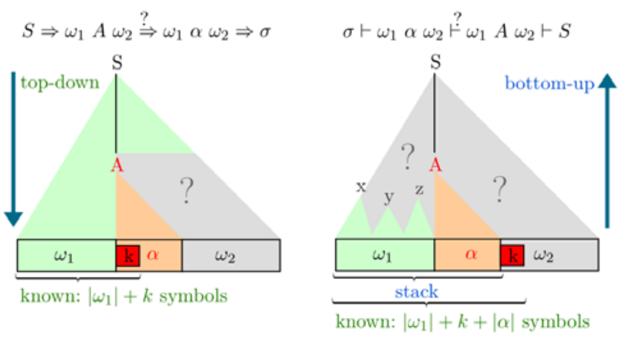

$S \Rightarrow \omega_1\ A\ \omega_2 \overset{?}{\Rightarrow} \omega_1\ \alpha\ \omega_2 \Rightarrow \sigma$
$\sigma \vdash \omega_1\ \alpha\ \omega_2 \overset{?}{\vdash} \omega_1\ A\ \omega_2 \vdash S$
S
top-down
A
?
ω_1
k
α
ω_2
known: $|\omega_1| + k$ symbols
S
bottom-up
?
A
?
x
y
z
ω_1
α
k
ω_2
stack
known: $|\omega_1| + k + |\alpha|$ symbols

Fig. 1: Power – LL(k) versus LR(k) [5].

Another important aspect is that semantic evaluation during top-down analysis has access to semantic attributes that have already been computed, visualized by one big green area in the syntax tree in the left side of Figure 1, whereas semantic evaluation during bottom-up analysis can only process synthesized attributes, visualized by several small green areas in the syntax tree in the right side of Figure 1.

Aho et al. in [1] pose the following key question "How can we handle L-attributed syntax driven definitions on LR grammars?" Their answer is: "Build the parse tree first and then perform the translation."

This leads to the central design decision for BoB (the same as taken for Smart): first of all, syntactic analysis must be performed in a bottom-up manner and afterwards, semantic analysis can take place in a top-down manner.

Figure 2 shows a general overview of the solution approach. BoB processes files with compiler descriptions (extension .ATG) written in Cocol4BoB using some predefined text files, so called frames (extension .frm), containing templates for the files to be generated. BoB generates input files for Flex (extension .l) and Bison (extension .yy), which are necessary to generate the scanners and parsers. This intermediate step (Flex and Bison) has the advantage that BoB does not have to generate complex scanners and complex bottom-up parsers itself, as Flex and Bison already provide well-tested ones. Additionally, BoB delivers semantic analyzers (in C++), which are compatible with the scanners and parsers generated by Flex and Bison.

The BoB-Flex&Bison toolchain generates compilers which process the input files in two phases:

1. lexical analysis and bottom-up syntax analysis and
2. top-down semantic evaluation.

But these two phases must be strictly separated. For data exchange, appropriate data structures are used, that store the necessary lexical and syntactical data. In case of lexical or syntactical errors in phase 1, the second phase can and must be omitted, of course.

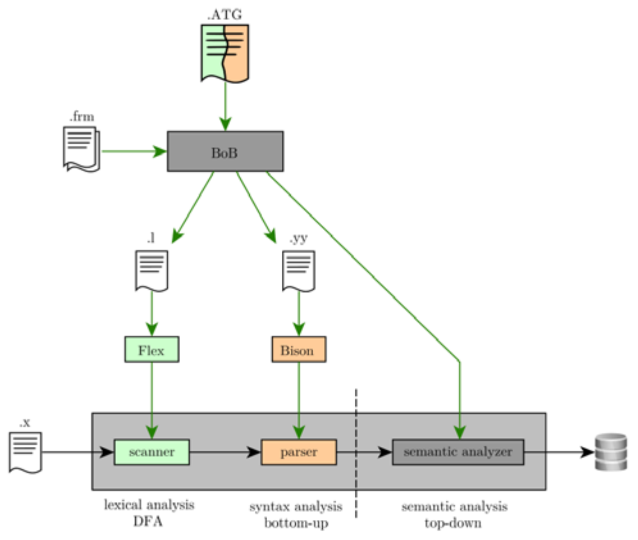


Fig. 2: General overview of the solution approach.

Another important aspect of Bob's design is that BoB itself is generated using Flex and Bison. The first version of BoB has been generated from hand-written .l and .yy files by Flex and Bison. Afterwards, an ATG for BoB in Cocol4BoB has been defined, which allows a full bootstrap: generating a new version of BoB with an older one, e.g., with the first version of BoB. If yet another pass of this bootstrapping process (e.g., generating the third version using the second one) still yields a functional version of BoB, it is shown, that all components of BoB are correct.

4.2 Requirements for and Sketch of Cocol4BoB

An important design decision concerns the definition of an adequate input language for BoB: a new compiler description language was developed that meets the following main requirements:

- User friendliness: high readability (e.g., with meaningful keywords) and only a few syntactical constructs for the whole compiler description (e.g., usage of EBNF for both, the description of the scanner and the parser, instead of regular expressions for the scanner and primitive BNF for the parser).
- Single source principle: only one input file should be necessary, which ensures consistency of the compiler description and avoids multiple declarations (with different notations).

According to these requirements, Cocol-2, the proven input language for Coco-2 [6], could be used as basis. But Cocol-2 could not be used without minor changes and a few extensions, because Cocol-2 uses symbols from the target language C++ (e.g., the operators << and >>) and BoB needs additional information which has to be incorporated into the language. Moreover, Flex and Bison do not support all concepts available in Cocol-2. Therefore, BoB uses a modified (e.g., <| and |> instead of << and >>) and extended (e.g., keyword *SEMDECL* or the *RECOVER BY* clause) version of Cocol-2 called *Cocol for Bob* or *Cocol4BoB* for short.

4.3 Architecture of BoB

The package and class diagrams in Figure 3 show the general architecture of Bob's object-oriented implementation in C++. BoB consists of the two static libraries *BoBTemplate* and *BoBCommon* and the executable called *BoB*:

- The library *BoBTemplate* contains functionality for the template mechanism used to generate files for scanners, parsers and semantic evaluators from corresponding templates provided in frame files (cf. Figure 2).
- The library *BoBCommon* contains the common parts of BoB (as well as for BoB-generated compilers) necessary for error handling in class *Errors* and some utilities in class *Utils*.
- The executable *BoB* consist of the class *BoB* which contains the main entry point. This class is responsible for handling user input/output and for initiating the scanner and the parser as well as the semantic evaluator. All other classes in this executable are generated by Flex and Bison from corresponding input files (either hand-written .l and .yy files or files generated by BoB from the BoB ATG by bootstrapping). The two classes *BoBScanner* and *BoBParser* obviously contain the generated scanner and the generated parser.

The library *BoBTemplate* is the centerpiece of BoB. It contains the all the algorithms used for the transformation from Bob's input in Cocol4BoB files to output files which subsequently serve as input files for Flex and Bison. The class *Template* is the main part of this library. This class provides an efficient *key-value store* which holds mappings for placeholders (*keys*) used in templates stored in frame files to target content (*values*) for the generated files. Based on this key-value store, *Template* provides functionality to fill a frame which contains the corresponding placeholders. The class *TemplateTokens* provides the definitions of the used frames and the used placeholders.

The classes *ScannerTemplate*, *ParserTemplate* and *SemEvalTemplate* differ in the particular target component. They contain special transformation methods, which take as input the data provided by the scanner and parser. These methods create the desired output files using the associated key-value store. Finally, after the whole input processing, the output files are generated by filling the frames.

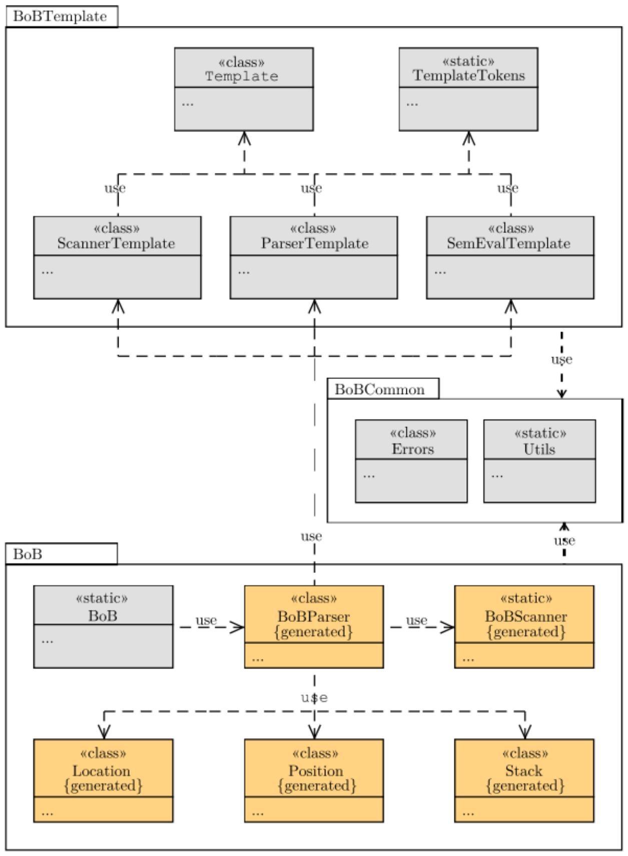


Fig. 3: Architecture of BoB.

4.4 Architecture of Compilers generated by BoB

The package and class diagrams in Figure 4 show the architecture of compilers generated by BoB (also compare this to Figure 3). The *X* in the diagrams stands for the name of the generated compiler (whose input language typically is called *X* as well). The main class *X* can either be derived from the provided frame file *Main.frm* (by copying *Main.frm* to *X.cpp* and applying some adaptations) or be implemented from scratch by the developer. The library *BoBCommon* is identical to the library used by *BoB* (cf. Figure 3) as described in the previous section. It contains error handling and some utility functionality.

The classes *XParser*, *XScanner*, *Location*, *Position* and *Stack* are generated by the *BoB-Flex&Bison* toolchain from an input file called *X.ATG* written in Cocol4BoB. These classes have exactly the same functionality as the corresponding classes within BoB (cf. Figure 3). The new class *XSemEval* (generated by BoB directly) contains the semantic evaluator. This class provides intermediate data structures, which at runtime will be filled by the scanner as well as the parser, and it contains all the semantic actions, which BoB copied from the input in *X.ATG* into this class. (This is a main difference in comparison to Smart, which uses special semantic code.) The semantic evaluation will be initiated by the main class *X*.

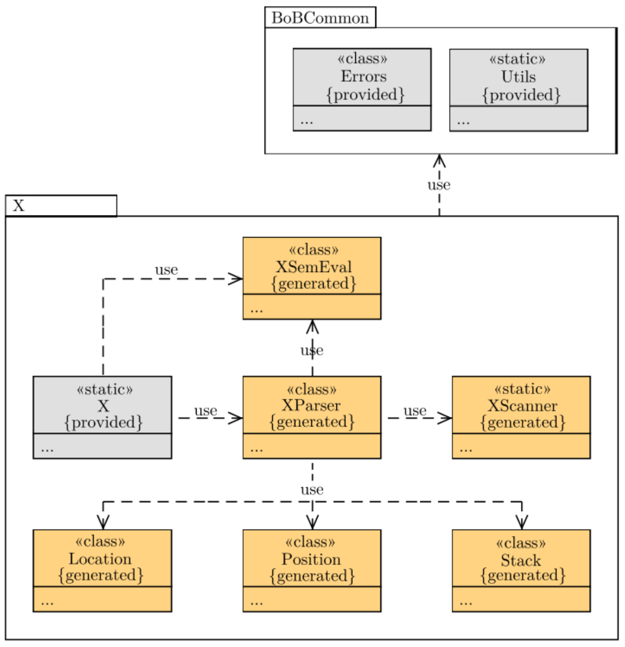

BoBCommon
«class» Errors {provided}
«static» Utils {provided}
use
X
«class» XSemEval {generated}
use
«static» X {provided}
«class» XParser {generated}
«static» XScanner {generated}
«class» Location {generated}
«class» Position {generated}
«class» Stack {generated}
...

Fig. 4: Architecture of compilers generated by *BoB*.

5. Implementation Aspects (for parser generation only)

BoB gets as input an ATG in Cocol4BoB, which describes the (main parts of the) compiler to be generated. BoB extracts the relevant information for the scanner, the parser as well as the semantic evaluator and uses a template mechanism to fill in the predefined frames. Figure 5 shows the relevant parts of the input file and the overall structure of the generated output files. It is apparent, that Flex and Bison use the information of the keywords, tokens and token classes corporately. Nevertheless, to avoid that the ATG has to be read and processed twice, the output frames are filled concurrently (within a single pass).

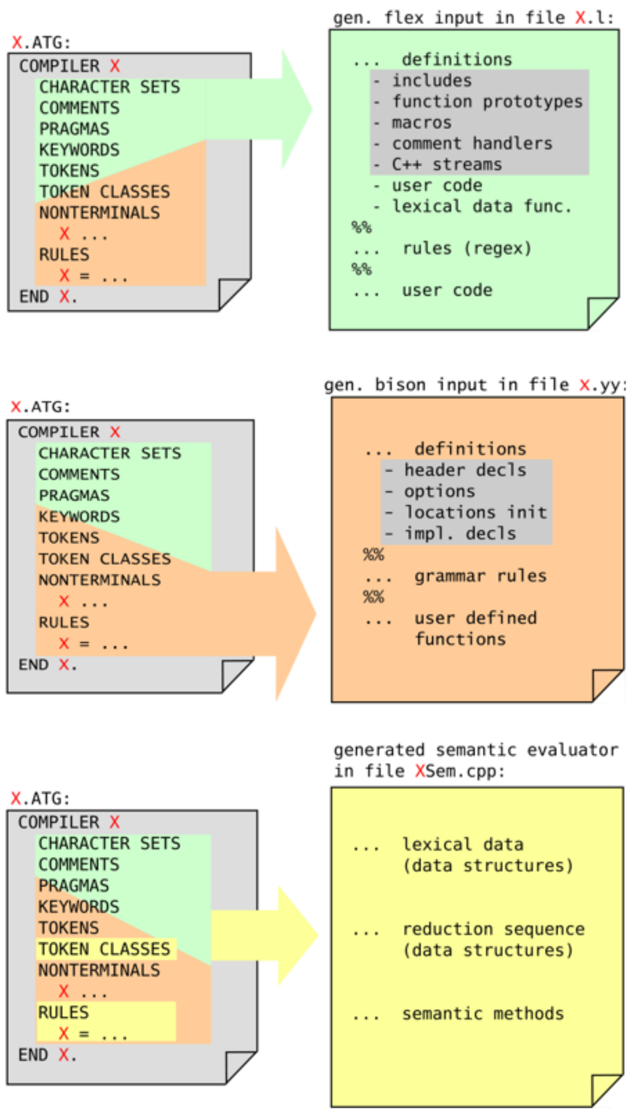


Fig. 5: Transformation of an ATG for language/compiler *X* in Cocol4BoB to input for Flex (*X.l*) and Bison (*X.yy*) as well as the semantic evaluator (*XSem.cpp*).

5.1 Grammar Rules – EBNF to BNF

The grammar rules in the ATG specify the syntactic structure that must be met by the input of the generated compilers. In Cocol4BoB these grammar rules may be defined using EBNF. But the generated rules in the target file (.y for Bison) have to be in BNF. Therefore, a transformation from EBNF to BNF is required. Mössenböck in [14] describes a method, which replaces the EBNF constructs for grouping (…), option […] and repetition {…} with artificial nonterminal symbols and corresponding BNF rules.

Table 1 demonstrates by use of simple examples, that there are (at least) two variants for applying these transformations. The difference between the two variants is, that

- variant 1 uses ε (empty sequence) rules whereas
- variant 2 uses additional alternatives.

So variant 1 leads to shorter rules and is therefore preferred. Unfortunately, the resulting ε alternatives also have a downside: in certain situations they lead to LR(1) conflicts (detected later on by Bison). As these conflicts can easily be resolved by hand, BoB nevertheless uses variant 1, because of the advantage of much shorter rules in realistic, more complex examples.

Tab. 1: Transformation of EBNF to BNF (based on [14])

EBNF	BNF	
	Variant 1	Variant 2
$A = a\ (b \mid c).$	$A \to a\ X$ $X \to b \mid c$	
$A = a\ [b].$	$A \to a\ X$ $X \to b \mid \varepsilon$	$A \to a \mid a\ X$ $X \to b$
$A = a\ \{b\}.$	$A \to a\ X$ $X \to X\ b \mid \varepsilon$	$A \to a \mid a\ X$ $X \to b \mid X\ b$

5.2 Semantic Evaluator

The semantic evaluator uses two data structures, containing

1. the lexical data (not the complete sequence of tokens but the values for token classes only, these values are provided by the scanner) and
2. the reduction sequence (defining the structure of the syntax tree, provided by the parser).

The main functional part of the semantic evaluator is comprised by the collection of the semantic actions copied form the ATG into semantic methods.

In Cocol4BoB there is only one kind of element which can provide lexical data: *token classes* (e.g., for numbers). Token classes represent terminal symbols with associated lexical data. Figure 6 (a) defines a simple grammar which consists of non-terminal symbols (in upper-case letters) and terminal symbols (tokens in lower-case letters). The terminal symbols have associated attributes (simple values, *val* for short), which provide lexical data for the semantic evaluator. These lexical values are needed by the semantic evaluator in the order of their occurrence in the input file. Figure 6 (b), as an example, shows the syntax tree for the input sequence *bdac*. The lexical data for this sequence is stored in a queue shown in Figure 6 (c), a *first-in-first-out* (FIFO) data structure (e.g., the number 1423 for the first element *b* of the sequence).

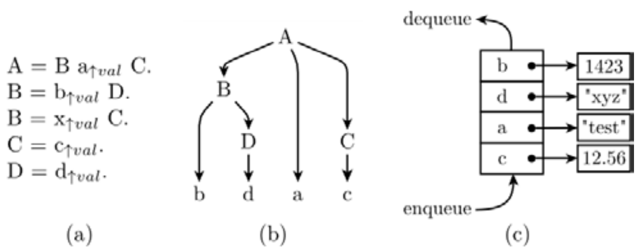
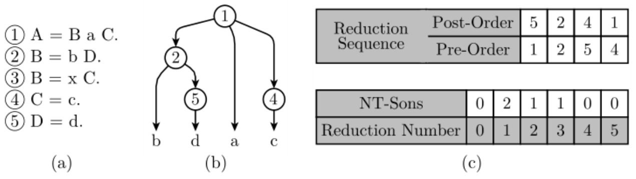


Fig. 6: Rules (a), syntax tree (b) and lexical data (c) based on [14].

To perform the semantic evaluation in top-down manner, the reduction sequence has to be captured during the bottom-up syntax analysis. The basic idea for this approach was first presented by Mössenböck in [14], Schmeiser and Barnard in [18] present a similar idea. BoB uses Mössenböck's idea and adds some modifications to meet the requirements of the semantic evaluator. While the reduction sequence defines a tree-based structure, Bob's implementation uses a linear data structure for its representation, a *double-ended queue* (dequeue, the sequential STL [20] container *std::deque*). This linear data structure has two advantages over a tree-structured one (like the associative STL container *std::set*):

1. memory allocation for such a linear data structure is more efficient, as only few but large blocks of memory must be allocated and
2. in total, a deque needs less memory than a tree.



Reduction Sequence						
	Post-Order	5	2	4	1	
	Pre-Order	1	2	5	4	

NT-Sons	0	2	1	1	0	0
Reduction Number	0	1	2	3	4	5

Fig. 7: Rules (a), syntax tree (b) and red. sequence (c) based on [14].

Figure 7 shows for an example the reduction sequence. An entry in the reduction sequence represents a rule in the grammar. To accomplish this, each rule in the grammar is associated with a number starting with 1. The number 0 has a special meaning: it represents a node with no sons, such nodes are usually used to exit loops. The reduction sequence is built by special semantic actions added to the rules of the generated parser. Because rules have a different number of non-terminals, nodes have a different number of sons. Therefore, it is necessary, to save the number of sons for each rule/node. This information is calculated by BoB during the creation of the compiler.

There is another problem with the reduction sequence described so far: Within a bottom-up parser, left recursion is preferred over right recursion, as left recursion saves space in the parser's stack. Therefore, BoB uses left recursion for transforming EBNF repetition {...} to BNF, see Figure 8 (a). But the reduction sequence created by left recursion, see Figure 8 (b), is different from the one created by right recursion, see Figure 8 (c). For the semantic evaluator, it is important, that the "loop reduction number" and the reduction numbers contained within the loop, are evaluated successively, because otherwise the processing order of the reduction numbers is disturbed. The correct order is guaranteed only by a reduction sequence created by right recursion, see Figure 8 (d). Therefore, a transformation from left to right recursion (only within loops) is required.

```
  EBNF          BNF (left recursion)        BNF (right recursion)

                Loop:                       Loop:
                    Loop_1;                     Loop_1;
Loop =          Loop_1:                     Loop_1:
  { Expr1           /* eps */        ⓪          /* eps */        ⓪
  | Expr2         | Loop_1 Expr1     ①        | Expr1 Loop_1     ①
  }.              | Loop_1 Expr2;    ②        | Expr2 Loop_1;    ②
```

(a)

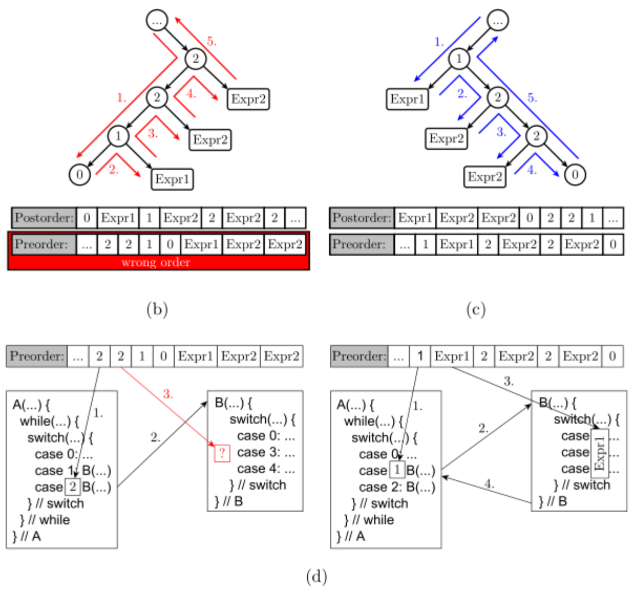


Fig. 8: Transformation of repetition from EBNF to BNF with left and right rec. (a), red. sequ. for loop created by left rec. (b) and by right rec. (c), processing of the red. sequ. created by left and right rec. – red. sequ. by left rec. is not processable by top-down semantic methods (d).

As shown in Figure 9, this transformation is divided into three steps: In the first step, all subsequent loop calls are added to a temporary buffer. All other nodes within the loop are added directly to the resulting reduction sequence. When the loop terminates, the next two steps are performed. In step 2, the first entry in the buffer is added to the reduction sequence. And finally, in step 3, the last elements in the buffer are added in reverse order to the reduction sequence.

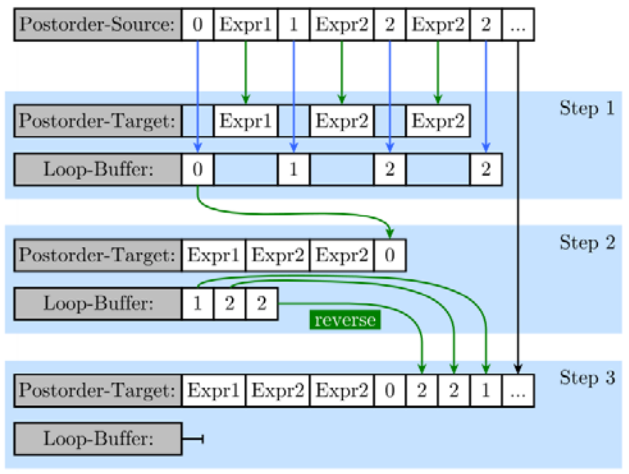


Fig. 9: Transformation of the red. sequ. for the example in Fig. 8 from left to right recursion.

For each grammar rule, BoB creates one semantic method which holds and executes all the semantic actions of this rule. The grammar rule for the root non-terminal (the first rule in a Cocol4BoB ATG) provides the main method for the semantic evaluation process. This method is called when the scanning and parsing of the input is finished. The structure of semantic methods is similar to recognition functions created for a compiler using the method of recursive descent [1]. The difference is that no lexical and syntactical actions are performed, as these have already been executed during the analysis phase. The following code snippet in Cocol4BoB shows an example EBNF grammar rule with the relevant parts for the semantic method:

```
A<|①|> =
        LOCAL<|②|>
        PRE  <|③|>
        POST <|④|>
        a<|⑥|> SEM<|⑤|>
        B<|⑦|> SEM<|⑤|>     ❶
      | ( b<|⑥|> ❺ )         ❷
      | [ C<|⑦|> ❻ ]         ❸
      | { c<|⑥|> ❼
        | d<|⑥|> ❽ }         ❹
.
```

The numbers ❶ – ❽ represent the reduction numbers for the resulting BNF grammar rules. And the numbers ① – ⑦ represent the relevant data which is extracted from the grammar rules for the creation of the semantic method (see code snippet in C++ below). The following list explains the data used:

① Formal parameter list of semantic method.
② Local semantic declarations.
③ Semantic action interleaved at the start of the method.
④ Semantic action interleaved at the end of the method.
⑤ Semantic action – before, after or between symbols.
⑥ Actual parameter list to get the lexical data.
⑦ Actual parameter list for calls of semantic methods.

The generated semantic method basically contains a *switch* statement to select the different alternatives. The control flow is defined by the reduction sequence. Within the *switch* condition, a call to the method *NextRedNr* delivers the next reduction number and selects the next alternative. Within an alternative, following components can occur:

- For each terminal class, a call to a *GETxAttr* method is inserted, this method delivers the captured lexical data.
- For each non-terminal symbol, a call to the associated semantic method is inserted.
- The semantic actions are copied directly from the grammar.
- A grouping construct (…) in EBNF is implemented by a *switch* statement.
- An option construct […] in EBNF is again implemented with a *switch* statement and has an additional ε alternative.
- The repetition {…} construct in EBNF is implemented like the option construct. Additionally, the construct is surrounded by a *while* statement, which enables repetition (without recursive calls). The ε alternative is used as exit point from the loop.

As can be seen for the artificial rules (grouping, option and repetition), no separate semantic methods are created. These constructs are implemented as embedded statements which act like separate methods. This has the advantage, that there are no scope problems with semantic variables. The code snippet in C++ below shows the semantic method generated from the grammar rule *A* shown above:

```
void A(①) {
      ② ③
      bool done = false;
      switch (NextRedNr()) {
        case ❶:
          GETaAttr(⑥); /*SEM*/ ⑤ /*SEM*/
          B(⑦);        /*SEM*/ ⑤ /*SEM*/
          break;
        case ❷:
          switch (NextRedNr()) {
            case ❺:
              GETbAttr(⑥);
              break;
          } // switch
          break;
        case ❸:
          switch (NextRedNr()) {
            case ❶:
              break;
            case ❻:
              C(⑦);
              break;
          } // switch
          break;
        case ❹:
          while (!done) {
            switch (NextRedNr()) {
              case ❶:
                done = true;
                break;
              case ❼:
                GETcAttr(⑥);
                break;
              case ❽:
                GETdAttr(⑥);
                break;
            } // switch
          } // while
          break;
      } // switch
      ④
} // A
```

6. Evaluation

The new compiler generator toolchain BoB-Flex&Bison combines the advantages of the simple input language for Coco-2, the powerful bottom-up syntax analysis of Bison and the powerful top-down semantic analysis concepts of Smart and Coco-2. With the implementation of BoB presented in this paper, the goals (see chapter 2) were reached and the requirements (see section 4.2) were met.

To assess especially the non-functional quality aspects of BoB, a comparison with existing compiler generators is necessary. Hence, the pros and cons of the (generated) compilers in terms of usability, capability and performance are compared.

6.1 Usability

The usability of a compiler generator heavily depends on the compiler description language, since this language is the "user interface" for a compiler generator. Coco-2 uses Cocol-2, which was designed for simplicity and usability. This is achieved primarily through the use of a single input file, meaningful keywords and the use of EBNF for both, the description of the scanner as well as the parser.

On the other hand, Flex and Bison use separate files for the description of the scanner and the parser. The scanner is described by regular expressions, while the parser is described by BNF rules. This results in the disadvantage that duplicate declarations are necessary in different notations, which can lead to inconsistency. Moreover, BNF is less expressive than EBNF used by Cocol-2.

BoB uses a modified and extended version of Cocol-2 called *Cocol4BoB*. This language includes adjustments to the target language C++ and to the compiler generators Flex and Bison. Thus, it can be stated that Cocol4BoB was built on an already established language and provides all its advantages.

Another aspect concerning the usability of BoB is the length of the toolchain: Since BoB is an additional component, it increases the complexity. This problem can be solved with automation by appropriate batch/shell scripts or by using a build tool like *CMake* (see www.cmake.org).

6.2 Capability

The capability of a compiler generator is defined by the power of the ATGs and the kind of supported attributes in semantic actions. Coco-2 supports LL(1) grammars only. LL(1) grammars are usually sufficient, but there are languages that cannot be described with this class of grammars. But Coco-2 supports powerful L attribution. Thus, inherited and synthesized attributes are allowed within semantic actions.

In contrast to Coco-2, Bison can handle LALR(1) grammars. LALR(1) grammars are a superset of LL(1) grammars and can therefore describe a much larger class of languages. On the other hand, Bison only supports the S attribution or a kind of LR attribution using global variables.

So, the presented compiler generators have their strengths and weaknesses regarding to their power in parsing and evaluation of semantics. BoB has set out to combine the strengths of them. Therefore, BoB supports L-attributed LALR(1) grammars. In this respect, BoB is superior to the other compiler generators mentioned here. Additionally, its relatively simple input language Cocol4BoB lowers the initial hurdles for aspiring compiler developers.

6.3 Performance

The advantages of BoB are not for free. A downside of BoB is the minor performance during creation of the compilers. But the main drawback is the performance of the generated compilers, which is significantly worse than those generated with Coco-2 or Flex and Bison.

The generation of compilers with BoB takes up to 20 % more time in comparison with Coco-2 as well as with Flex and Bison. Moreover and most important, compilers generated by Flex and Bison are up to six times faster, compilers generated by Coco-2 are up to 30 % faster than those generated by BoB. Until now, no optimizations with respect to the runtime of BoB-generated compilers have been applied, this is left to further work, and we expect substantial speedups, so that BoB-generated compilers can cope with compilers generated by Coco-2, but the efficiency of compilers generated by Flex and Bison is out of reach.

Another aspect to be considered is memory usage: Whereas compilers generated by Coco-2 or by Flex and Bison have constant memory complexity in the length of the input sequence, those generated by BoB have a linear complexity (mainly because of the reduction sequence).

So compiler developers have to decide which characteristics are more important for their applications. There are applications where the benefits of BoB exceed the drawbacks in runtime and memory usage, especially for educational purposes.

7. Conclusions

In this paper we introduced the new compiler generator BoB, implemented as preprocessor for Flex and Bison. BoB not only combines the strengths of bottom-up syntax analysis (LR grammars) and top-down evaluation of semantics (L-attribution), but it also has a simple to use and consistent input language for compiler description.

Since BoB (using Flex and Bison) generates compilers in C++, it is restricted to this target language. Because BoB supports the powerful L-attributed LALR(1) grammars, the compiler developers have not to care for LL(1) conflicts and can use the intuitive inherited and synthesized attributes. Therefore, BoB is the ideal compiler generator toolchain for beginners. More details especially for implementation aspects of scanner generation can be found in [4].

Appendix

In the following, we present a small but complete example written in Cocol4BoB for the evaluation of simple arithmetic expressions (basic operations on integers only), comparable to the task of a calculator (e.g., 17 + 4 = 21).

The same example is used for Coco-2 in [6] and [7], but the notation there is Cocol-2 so Coco4BoB can be compared to Cocol-2.

```
COMPILER Calc

CHARACTER SETS
  Digit      = '0' .. '9'.
  whiteSpace = CHR(9) + EOL IGNORE.
    /*tab and end of line, blank ignored by default*/

COMMENTS
  FROM '--' TO EOL.           --Ada comments
  FROM '//' TO EOL.           //C++ comments
  FROM '/*' TO '*/'.          /*C comments*/
  FROM '(*' TO '*)' NESTED.   (*Modula-2 comments*)

TOKENS
  '+'. '-'.
  '*'. '/'.
  '('. ')'.

TOKEN CLASSES
  number<|int& val|> =
    digit { digit }   LEX <|val = atoi(tokenStr);|>.

NONTERMINALS
  Calc.
  Expr<|int &e|>.
  Term<|int &f|>.
  Fact<|int &t|>.

RULES
Calc =                LOCAL<|int e = 0;|>
  Expr<|e|>           SEM<|cout << " = " << e;|>.

Expr<|int &e|> =      LOCAL<|int t = 0;|>
  Term<|e|>
  { '+' Term<|t|>     SEM<|e = e + t;|>
  | '-' Term<|t|>     SEM<|e = e - t;|>
  }.

Term<|int &t|> =      LOCAL<|int f = 0;|>
  Fact<|t|>
  { '*' Fact<|f|>     SEM<|t = t * f;|>
  | '/' Fact<|f|>     SEM<|t = t / f;|>
  }.

Fact<|int &f|> =
  number<|f|>
  | '(' Expr<|f|> ')'.

END Calc.
```

References

[1] Aho, A. V., M. S. Lam, R. Sethi and J. D. Ullman: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[2] Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A Nelson, D. Syre, P. B. Sheridan H. Stern, I. Ziller, R. A Hughes and R. Nutt: *The Fortran automatic coding system*. Proceedings of the IRE-AIEE-ACM, 1957.

[3] Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger: *Revised report on the algorithm language Algol 60*. Communications of the ACM, 6(1), 1963.

[4] Dichler, W.: *BoB: Best of Both in Compiler Construction – Combination of Bottom-up Syntax Analysis and Top-down Semantic Evaluation.* Master Thesis at the University of Applied Sciences Upper Austria, Hagenberg, 2013.

[5] Dobler, H.: *Formal Languages and Automata Theory* (in German). Technical Report, University of Applied Sciences Upper Austria, Hagenberg, 2011.

[6] Dobler, H. and K. Pirklbauer: *Coco-2: A New Compiler Compiler*. ACM SIGPLAN Notices, Vol. 25, No. 5, 1990.

[7] Dobler H.: *A Hybrid Top-Down Parsing Technique*. ACM SIGPLAN Notices, Vol. 26, No. 3, 1991.

[8] Donnelly, C. and R. Stallman: *Bison – The Yacc compatible Parser Generator*. Free Software Foundation, Inc., 2.5 Edition, 2011.

[9] ISO: *Information Technology – Syntactic Metalanguage – Extended BNF*. Norm ISO/IEC 14977:1996(E), International Organization for Standardization, 1996.

[10] Johnson, S. C.: *Yacc: Yet Another Compiler-Compiler*. Technical Report, Bell Laboratories, 1975.

[11] Katwik, J. van: *A preprocessor for Yacc or A poor man's approach for parsing attributed grammars*. SIGPLAN Notices Vol. 18, No. 10, October 10th, 1983.

[12] Knuth, D.: *Semantics of context-free languages*. Mathematical Systems Theory, 1968.

[13] Lesk, M. E. and E. Schmidt: *Lex – A Lexical Analyzer Generator*. Technical Report, Bell Laboratories, 1975.

[14] Mössenböck, H.: *Compiler Generating Systems for Microcomputers* (in German). PhD Thesis at the Johannes Kepler University Linz, Verlag VWGÖ, Wien, 1987.

[15] Mössenböck, H.: *The Compiler Generator Coco/R*. 2011. www.ssw.uni-linz.ac.at/Research/Projects/Coco

[16] Paxson, V., W. Estes and J. Millaway: *flex*. The flex Project, 2.5.35 Edition, 2007.

[17] Rechenberg, P. and H. Mössenböck: *A Complier Generator for Microcomputers*. Hanser, 1989.

[18] Schmeiser, J. P. and D. P. Barnard: *Producing a top-down parse order with bottom-up parsing*. Information Processing Letters 54, 1995.

[19] Sippu, S. and E. Soisalon-Soininen: *Parsing Theory – Volume II: LR(k) and LL(k) Parsing*. EATCS Monographs on Theoretical Computer Sciences: European Association for Theoretical Computer Science. Springer, 1990.

[20] Stroustrup, B.: *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 2013.

[21] TIOBE Software: *TIOBE Programming Community Index*, 2013. www.tiobe.com/tpci.htm.

[22] Wirth, N.: *What can we do about the unnecessary diversity of notation for syntactic definitions?* Communications of the ACM, 20(11), 1977.

[23] Wirth, N.: *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 3rd edition, 1985.

Wolfgang Dichler BSc (2011), MSc (2013) in Software Engineering, Software Developer for Daimler and Magna Power Train, Software Architect at MP2 IT Solutions.

Heinz Dobler Dipl.-Ing. in Computer Science (1986) Dr. tech. (1993), Assistant Professor at Johannes Kepler University Linz, Austria; Professor for Software Engineering at the University of Applied Sciences Upper Austria; Member of the ACM and IEEE.