Is it BabyCobol-proof?

Virtual Meetup on Language Engineering

Dr. Vadim Zaytsev aka @grammarware, 15 April 2021

IDENTIFICATION DIVISION.

- Vadim Zaytsev aka @grammarware ( )
  - research (, , CWI)
  - teaching ( )
  - industry (RAINCODE mainframe to .NET, raincode LABS)
- Interests:
  - language engineering
  - legacy systems
  - reliable grammarware

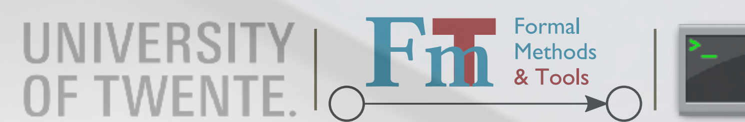
http://grammarware.net && http://grammarware.github.io

- We teach:
  - from scratch development
  - projects up to 10kLOC
  - clear problems
  - choice among good options
  - delivery ≡ retirement
- They get:
  - maintenance
  - support in production
  - uncomfortable requests
  - legacy code & documentation

The World Runs on Legacy

- 43% of banking systems built on COBOL [R]
- 15% of new applications built in COBOL [G]
- 75% of business data processed by COBOL [G]
- 80% of in-person transactions run COBOL [R]
- 95% ATM swipes rely on COBOL code [R]
- 180-200[G], 220[R] billion LOC in use
  - one codebase up to 250 MLOC [Bankia]
- Reminder: PL/I, HLASM, 4GLs, ...

[G] Gartner 2003, [R] Reuters 2017



BabyCobol

- One language
  - to condense SLE’s worst fears
- Small enough to be implementable
  - within one paper
  - or one course
- Each language feature represents
  - one actual problem
  - from a 2GL/3GL/4GL
  - a lot of interplay
- ≠ BF, ≠ INTERCAL, ⊄ COBOL

Zaytsev, *Software Language Engineers' Worst Nightmare*, SLE’20

BabyCobol: The Language Reference

BabyCobol is a project in language design aimed at creating a language that is, on one hand, small enough to be quickly implementable (fully or partially) within any framew
support its features, and, on the other hand, complex enough to cover typical problems of legacy language processing. If you learn how to compile MiniJava, you stand a go
implementing a reasonably good compiler for any contemporary programming language. If you show how your language extension works on Featherweight Java, it has a go
being applicable to any reasonable modern object-oriented programming language. If you can handle BabyCobol with your tool and with your skills, you are ready to face th
software modernisation, codebase migration and legacy language processing in general. At this day and age, being future proof means being able to handle software of the p

Features:

Origins

AppBuilder | CLIST | COBOL | FORTRAN | PL/I | REXX | RPG

Mentions

- UTwente EEMCS Faculty, Formal Methods and Tools, *Manfred Paul Award for Vadim Zaytsev*, a news item, 3 February 2021
- IFIP TC2, *Manfred Paul Award for Excellence in Software Theory and Practice* "for boldness in seeking real-world test cases for modern software language engineering tools by mining languages from the distant past", 22 December 2020
- Vadim Zaytsev, *Software Language Engineers' Worst Nightmare*, a paper published at SLE 2020, doi:10.1145/3426425.3426933, 15–16 November 2020
- Vadim Zaytsev, *Software Language Engineers' Worst Nightmare*, a pre-recorded presentation at SLE@SPLASH, 13 November 2020
- Bernd Fischer, *Breaking Parsers: Mutation-based Generation of Programs with Guaranteed Syntax Errors*, IFIP TC-2 WG 2.11 on Program Generation, invited/impromptu presentation, Sorbonne Université, 20 February 2020 (first implementation of BabyCobol in Prolog capable of generating hundreds of BabyCobol programs)
- Vadim Zaytsev, *BabyCobol: The Challenge to Program Generation Tool Developers*, IFIP TC-2 WG 2.11 on Program Generation, invited presentation, Sorbonne Université, 17 February 2020
- Vadim Zaytsev, *Legacy and Software Renovation*, Software Evolution guest lecture, Universiteit van Amsterdam, 9 December 2019
- Vadim Zaytsev, *Blind Men and a Room Full of Elephants*, BENEVOL 2019 keynote, Vrije Universiteit Brussel (VUB), 28 November 2019

https://slebok.github.io/baby

BabyCobol is a project by Dr. Vadim Zaytsev a.k.a. @grammarware. Page last updated in March 2021.

W3C XHTML 1.1 W3C CSS 3

BabyCobol in a Nutshell

- ADD/SUBTRACT/MULTIPLY/DIVIDE + EVALUATE
- ACCEPT/DISPLAY + PICTURE
- MOVE (CORRESPONDING)
- GO TO + ALTER + PERFORM
- Divisions, paragraphs, sentences, statements
- Expressions can be contracted
- Qualification needs to be sufficient

Lexical Analysis

IF MOD-G-JAHR NOT NUMERIC MOVE ZERO TO MOD-G-JAHR.

Rainer Gerhards, CC-BY-SA, 2006, Punch-card-cobol

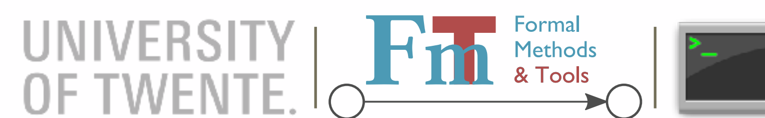
Lexical Analysis

- Indentation is punchcard-driven
  - parsing starts at X column
  - line continuation markers
- Whitespace is insignificant
  - but used for disambiguation
- Keywords ≠ reserved
  - can be used as field names
- Lexical imports
  - COPY A REPLACING ===B=== WITH ===C===

```
FUNC : PROC OPTIONS(MAIN);
 DCL IF   FIXED DEC(3,0) INIT (1);
 DCL THEN FIXED DEC(3,0) INIT (1);
 DCL ELSE FIXED DEC(3,0) INIT (1);

IF   IF   = ELSE
THEN THEN = IF
ELSE ELSE = THEN;

DISPLAY( IF  );
DISPLAY( THEN );
DISPLAY( ELSE );
END;
```

Syntactic Analysis

- IF X > 100 OR Y < 10 THEN STOP.
- IF X > 100 OR < 10 THEN STOP.
- IF X = 100 OR 10 THEN STOP.
- IF X > 100 OR < 10 AND NOT 0 THEN STOP.
- IF X NOT = 10 OR 100 THEN STOP.
- IF (X < 10 OR > 100) AND 0 THEN STOP.
- IF X NOT = 42 AND (Y < 10 OR > 100) AND 0 THEN STOP.
- …



GO TO Considered Harmful?

Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of (*A*1, *A*2, ···, *An*)), or conditional expressions as introduced by J. McCarthy (*B*1 → *E*1, *B*2 → *E*2, ···, *Bn* → *En*), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

A Case against the GO TO [EWD215]

EWD215 - 0

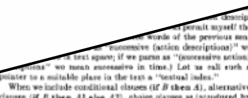
A Case against the GO TO Statement.

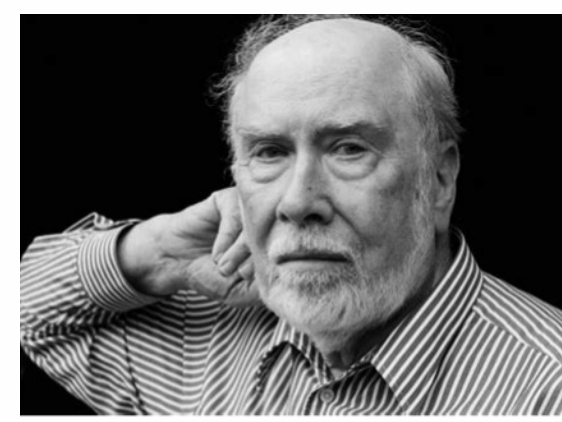
by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to effectuate the desired effect; it is this process that in its dynamic behaviour has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost best to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.





Dynamic Semantics

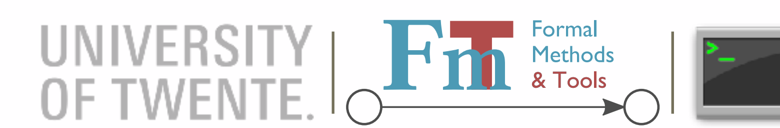
- GO TO
  - considered harmful
- ALTER?
  - not considered

Static GO TO

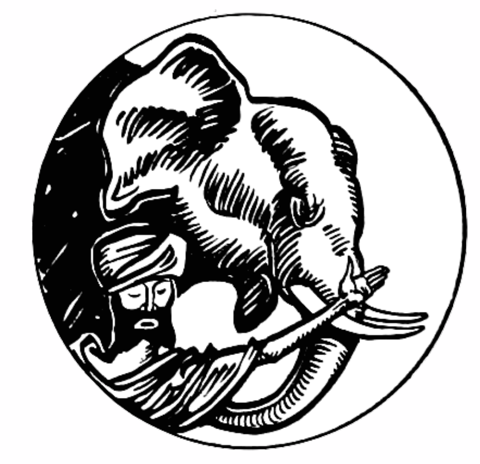
```
______ PROCEDURE DIVISION.

           . . .

       EXIT-ON-ERROR.
           GO TO EXIT-UPDATE-RECORD.
       EXIT-UPDATE-RECORD.
           . . .
       EXIT-ROLLBACK-RECORD.
           . . .
```

Alterable GO TO

```
______ PROCEDURE DIVISION.
           . . .
       ALTER EXIT-ON-ERROR TO PROCEED TO EXIT-ROLLBACK-RECORD.
           . . .
       EXIT-ON-ERROR.
           GO TO EXIT-UPDATE-RECORD.
       EXIT-UPDATE-RECORD.
           . . .
       EXIT-ROLLBACK-RECORD.
           . . .
```



Computable GO TO



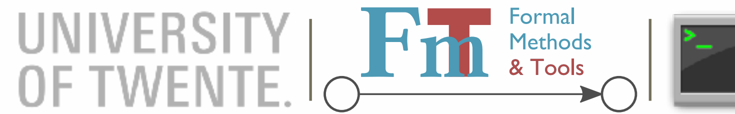
```
______ PROCEDURE DIVISION.
           . . .
       MOVE "EXIT-ROLLBACK-RECORD" TO HANDLER.
           . . .
       EXIT-ON-ERROR.
          GO TO HANDLER.
       EXIT-UPDATE-RECORD.
           . . .
       EXIT-ROLLBACK-RECORD.
           . . .
```

Summary of BabyCobol

- Hard to parse
  - indentation, whitespace, keywords, contractions, …
- Hard to analyse
  - name-based assignment, sufficient qualification, …
- Hard to execute
  - control flow self-modification
- Still easier to handle than legacy

Software Evolution: Retrospective

- As an idea, it works
  - minor detail can be polished
- Narrative ≠ Complexity
  - plot thickens differently than challenges
- Gamification led to a product line
  - focus: data types vs control structures
- Series of small tools
  - instead of one compiler/interpreter



Conclusion

- Legacy is a scary world
  - that runs/supports your life
- Legacy management is teachable
- Are *you* BabyCOBOL-proof?
- Is your *toolset* BabyCOBOL-proof?
- Follow @grammarware!
- Questions?

Software Language Engineers' Worst Nightmare

Vadim Zaytsev
Universiteit Twente
Enschede, The Netherlands
vadim@grammarware.net

Abstract

Many techniques in software language engineering get their first validation by being prototyped to work on one particular language such as Java, Scala, Scheme, or ML, or a subset of such a language. Claims of their generalisability, as well as discussion on potential threats to their external validity, are often based on authors' ad hoc understanding of the world outside their usual comfort zone. To facilitate and simplify such discussions by providing a solid measurable ground, we propose a language called BabyCobol, which was specifically designed to contain features that turn processing legacy programming languages such as COBOL, FORTRAN, PL/I, REXX, CLIST, and 4GLs (fourth generation languages), into such a challenge. The language is minimal by design so that it can help to quickly find weaknesses in frameworks making them inapplicable to dealing with legacy software. However, applying new techniques of software language engineering and reverse engineering to such a small language will not be too tedious and overwhelming. BabyCobol was designed in collaboration with industrial compiler developers by systematically traversing features of several second, third and fourth generation languages to identify the core culprits in making development of compiler for legacy languages difficult.

CCS Concepts: • **Software and its engineering** → **Specialized application languages**; **Compilers**; • **Social and professional topics** → *Software maintenance.*

Keywords: domain-specific languages, legacy software, language engineering, software migration, teaching SLE

ACM Reference Format:
Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3426425.3426933

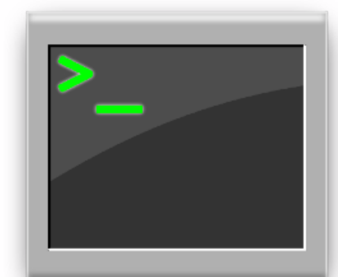
SLE '20, November 16–17, 2020, Virtual, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA*, https://doi.org/10.1145/3426425.3426933.

1 Introduction

Legacy languages designed in the second half of the last century, are still dominating some domains like the financial sector, and have ample presence in other highly critical domains such as insurance, logistics, manufacturing and military. Even in the programming community index TIOBE [68] languages like COBOL (#27), FORTRAN (#30) and RPG (#38) are constantly looming next to modern freshly designed and regularly updated languages like Dart (#26), Scala (#29) and Kotlin (#35). Only a small fraction of the users of such languages are happy customers deliberately making this technological choice for its actual benefits, the rest are forced by circumstances into maintaining business-critical systems that are too large and complicated to replace, rewrite or even re-engineer. Many owners of such legacy codebases invest substantially into their renovation, be it replatforming, rearchitecting, reverse engineering, language migration or anything else that is still a viable option for them.

Developers of compilers, debuggers, development environments, program restructuring tools, fact extractors, testing automation frameworks, etc, need to be ready to tackle all kinds of challenges posed by legacy languages. Yet, such challenges often remain some sort of sacred knowledge for developers with intimate familiarity with said legacy languages. Many new techniques are being proposed and published, targeting languages for which it is much easier to find enough open source code for experimenting, enough documentation for comprehension, and enough freely available base compilers to extend or compare to. With this project, we would like to bridge the gap by providing a description for a lab-made language that exemplifies an entire collection of issues that make it so challenging to tackle legacy languages. Inspired by languages like MiniJava [5] and Featherweight Java [33], that are extremely useful for academic researchers to apply their knowledge and techniques on (see § 2 for a more detailed treatment of related work), we are proposing a new language called **BabyCobol**. Unlike the infamous INTERCAL, standing for *Compiler Language With No Pronounceable Acronym*, which was specifically designed to have "nothing at all in common with any other major language" [76] and hence falls into the category of uselessly esoteric languages, BabyCobol was designed to exhibit most of the *actual real life* problems encountered in compiling and migrating legacy languages.

We present the language design across the next few sections, and then return to the real legacy languages driving



[doi:10.1145/3426425.3426933] [SLEBoK:baby]

________ THE END.

STOP.

* FOLLOW @GRAMMARWARE