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

SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11…$15.00
https://doi.org/10.1145/3426425.3426933

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

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 that design in § 6, to reflect in detail which of those languages inspired which BabyCobol features. § 7 concludes the paper.

To exemplify the look and feel of BabyCobol and to provide a brief reference to most of its features, we include Listing 1 with a complete program written in it. The corresponding commentary explains which features are used and refer to respective sections about them.

A complete and growing language reference of BabyCobol can be found at **https://slebok.github.io/baby**. Its design goals are twofold: to keep the language as small as possible, and at the same time to insert as many legacy language features into it without drastically abstracting from the details that make them hard to implement. Following the standard sane principles of good design, we also try to make it modular (so that it is sensible to implement only `ALTER` without `LOOP` or only line continuations without figurative constants), but this always has lower priority since sometimes the hardships manifest only as feature interactions (like the interaction between `GO TO` and `PERFORM THROUGH`).

2 Related Work

MiniJava [5] is a subset of Sun/Oracle's Java programming language [29], restricted for the purposes of teaching compiler construction. It is much smaller than actual Java, but complex enough to expose a number of problems that a compiler engineer should be familiar with. As such, it is very similar in both objectives and spirit to BabyCobol. The language is rather successful and has gained popularity among lecturers teaching compiler construction courses, especially those based on the book of Appel and Palsberg. The conceptual predecessors of MiniJava are numerous versions of nameless Pascal subsets used by compiler construction professors throughout the 1990s worldwide.

The popularity of Appel and Palsberg's MiniJava now overshadowed an earlier initiative with exactly the same name which developed a simplified extension of a subset of Java for teaching introductory programming [55]. There has also at some point been a similar COBOL-centred initiative called **Mini-COBOL** [27], published in 1969 as a subset of COBOL suitable for teaching students programming — its main design objective was to cover a subset that had the same behaviour across all available compilers (there are apparently around 300 dialects of COBOL in use [41]). Either of these has a goal drastically different from ours, so we list them here just to avoid confusion caused by their names. Conceptually these languages are closer to BASIC [37], ABC [26], Logo [1], perhaps also Scratch [46] and Racket [22].

Featherweight Java [33] is a minimal core calculus for Java, proposed to facilitate understanding the consequences of extensions and variations, proposed almost two decades ago. Featherweight Java has received substantial attention since then, with a definition of its denotational semantics [65], small-step operation semantics [23], type-preserving compilation [42], received proposed extensions like *Generic FJ* [79], *Feature FJ* [4], *Corecursive FJ* [2], *Transactional FJ* [70], *FeatherTrait* [44], *Welterweight Java* [52], *Middleweight Java* [9], *Featherweight Wrap Java* [8], and perhaps others.

If implementations of BabyCobol or applications of certain techniques to it can be made measurable in a comparable way, then we are looking at something approaching the famous **DeCapo** benchmarking suite [10]. The point of departure of the original DeCapo paper was similar to ours: Blackburn et al. were concerned with ungrounded generalisations of techniques developed for C and FORTRAN, with respect to their applicability to Java, and now we are concerned about applicability of methodologies developed for modern well-designed languages, to legacy languages. We do not have the ambition to propose benchmarks that can mean for the SLE community what POPLMark [6] meant for the PLT community, in that sense it falls somewhere between POPLMark and the expression problem[1] [69, 72], or its modern SLE reformulation [43]: an apparent inability of a given technology to express BabyCobol definitely sends a strong signal about its inapplicability to problems in legacy software codebases, and a successful coverage allows for detailed investigation of the solution to see how much boilerplate had to be produced and how many corners cut.

3 Syntax

First and foremost, let us focus on the character-level organisation of a typical legacy program.

3.1 Position-Based Syntax (and Semantics)

Indentation-aware languages such as Python [71], Occam [49], Haskell [30] or Kotlin [12] are quite popular and not as shunned now as they were some years ago [74]. However, in the very beginning of the programming language evolution assigning meaning to input characters based on which position within the string they were found, was rather common — examples include COBOL [61], HLASM [57] or pre-f90 FORTRAN [7]. For example, in COBOL, a non-space character in column 7 had special meaning, the most common being `*` or `/` signalling a comment, or a space signalling a line of executable code. Columns 1–6 denote a "sequence number", which remains largely unused nowadays, since it outlived its

[1] We note here that the expression problem is a well-defined symptom of a much deeper problem in data abstraction with the gap between the abstract data types and object-oriented programming [16–18]

```
(1)  ␣␣␣␣␣␣*␣Example program
(2)  ␣␣␣␣␣␣␣␣IDENTIFICATION DIVISION.
(3)  ␣␣␣␣␣␣␣␣␣␣␣␣PROGRAM-ID. FIB.
(4)  ␣␣␣␣␣␣␣␣␣␣␣␣AUTHOR. Anonymous.
(5)  ␣␣␣␣␣␣␣␣␣␣␣␣DATE-WRITTEN. 2019-11-01.
(6)  ␣␣␣␣␣␣␣␣DATA DIVISION.
(7)  ␣␣␣␣␣␣␣␣01 WORKING-STORAGE-AREA.
(8)  ␣␣␣␣␣␣␣␣␣␣␣␣05 END PICTURE IS 99.
(9)  ␣␣␣␣␣␣␣␣␣␣␣␣05 CUR PICTURE IS 9(20).
(10) ␣␣␣␣␣␣␣␣␣␣␣␣05 LAST.
(11) ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣09 N LIKE CUR.
(12) ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣09 N-1 LIKE CUR.
(13) ␣␣␣␣␣␣␣␣PROCEDURE DIVISION.
(14) ␣␣␣␣␣␣␣␣␣␣␣␣DISPLAY ENTER THE LIMIT
(15) ␣␣␣␣␣␣␣␣␣␣␣␣ACCEPT END.
(16) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE SPACES TO cur last.
(17) ␣␣␣␣␣␣␣␣␣␣␣␣DISPLAY N-1.
(18) ␣␣␣␣␣␣␣␣␣␣␣␣SUBTRACT 1 FROM END.
(19) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE 1 TO N.
(20) ␣␣␣␣␣␣␣␣␣␣␣␣PERFORM PRINT-FIB END TIMES.
(21) ␣␣␣␣␣␣␣␣␣␣␣␣STOP.
(22) ␣␣␣␣␣␣␣␣PRINT-FIB.
(23) ␣␣␣␣␣␣␣␣␣␣␣␣DISPLAY N.
(24) ␣␣␣␣␣␣␣␣␣␣␣␣ADD N-1 TO N GIVING CUR.
(25) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE NTON -1
(26) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE CURTON.
```

```
(27) ␣␣␣␣␣␣*␣Expanded program
(28) ␣␣␣␣␣␣␣␣IDENTIFICATION DIVISION.
(29) ␣␣␣␣␣␣␣␣␣␣␣␣PROGRAM-ID. FIB.
(30) ␣␣␣␣␣␣␣␣␣␣␣␣AUTHOR. Anonymous.
(31) ␣␣␣␣␣␣␣␣␣␣␣␣DATE-WRITTEN. 2019-11-01.
(32) ␣␣␣␣␣␣␣␣DATA DIVISION.
(33) ␣␣␣␣␣␣␣␣01 WORKING-STORAGE-AREA.
(34) ␣␣␣␣␣␣␣␣␣␣␣␣05 LIM PICTURE IS 99.
(35) ␣␣␣␣␣␣␣␣␣␣␣␣05 CUR PICTURE IS 99999999999999999999.
(36) ␣␣␣␣␣␣␣␣␣␣␣␣05 LAST.
(37) ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣09 N PICTURE IS 99999999999999999999.
(38) ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣09 N1 PICTURE IS 99999999999999999999.
(39) ␣␣␣␣␣␣␣␣PROCEDURE DIVISION.
(40) ␣␣␣␣␣␣␣␣␣␣␣␣DISPLAY "ENTER THE LIMIT".
(41) ␣␣␣␣␣␣␣␣␣␣␣␣ACCEPT LIM.
(42) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE SPACES TO CUR. MOVE SPACES TO LAST.
(43) ␣␣␣␣␣␣␣␣␣␣␣␣DISPLAY N1 OF LAST.
(44) ␣␣␣␣␣␣␣␣␣␣␣␣SUBTRACT 1 FROM LIM.
(45) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE 1 TO N OF LAST.
(46) ␣␣␣␣␣␣␣␣␣␣␣␣PERFORM PRINT-FIB LIM TIMES.
(47) ␣␣␣␣␣␣␣␣␣␣␣␣STOP.
(48) ␣␣␣␣␣␣␣␣PRINT-FIB.
(49) ␣␣␣␣␣␣␣␣␣␣␣␣DISPLAY N OF LAST.
(50) ␣␣␣␣␣␣␣␣␣␣␣␣ADD N1 OF LAST TO N OF LAST GIVING CUR.
(51) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE N OF LAST TO N1 OF LAST.
(52) ␣␣␣␣␣␣␣␣␣␣␣␣MOVE CUR TO N OF LAST.
```

Listing 1. Fibonacci numbers calculated in BabyCobol. The version on the left is written to make most use of problematic features of BabyCobol, while the version on the right avoids some of them and is thus easier to parse, compile and analyse, while being functionally equivalent. Lines (1) and (27) contain a comment (§ 3.1), lines (2), (6), (13), (28), (32) and (39) start separate divisions (§ 3.7), lines (8–9) and (34–38) contain picture clauses (§ 4.1), lines (11–12) — like clauses (§ 4.1), line (8) defines an identifier with a name equal to a keyword (§ 3.4), lines (12) and (22) declare an identifier and a paragraph with a dash in their names, line (14) does not use quotes and hence uses default values of three undefined fields (§ 4.4), line (16) relies on case insensitivity (§ 3.3), lines (15) and (51) execute picture-driven inputs, lines (17), (23), (43) and (49) — picture-driven output, line (16) uses a figurative constant on targets of different types (§ 4.3), lines (20) and (46) contain an out-of-line `PERFORM` statement (§ 5.2) calling the paragraph on lines (22–26) or (48–52), lines (25–26) exploit whitespace insignificance (§ 3.5), lines (17), (19), (23–26) use unambiguous insufficient qualification (§ 4.2).

usefulness together with the punch card sorting machines. In FORTRAN, comments were denoted as the *first* character of the line being equal to `C`, while line continuations were affected by column 6.

For BabyCobol, we mostly follow COBOL conventions: we skip over the first 6 columns, then treat column 7 as the indicator of the line status, column 8 serves for starting top level constructs (divisions, paragraphs, sections — see § 3.7), columns 12–72 are for regular code and line contents beyond column 73 are ignored (can be used for short comments, debug info, etc). Since essentially this makes the language a mix of a pattern language [3, 77] and a context-free language, even this feature alone probably already forces the person implementing a language processor to insert a manually written purely lexical preprocessor into their pipeline before the classic scanner or tokeniser.

3.2 Line Continuations

The significance of columns within the line adds a level of complexity for compiler writers because essentially it leads to the necessity of adding an extra component before tokenising in the beginning of lexical analysis or for fine-grained postprocessing pretty-printed code if the language processing pipeline involves unparsing. However, to truly experience these hardships, we will need to add line continuations into the mix.

The basic idea behind line continuations is simple and straightforward: whenever a line of code becomes too long (to fit on a punch card), there is a lexical mechanism supported by all language tools, to spread the contents of desired/conceptual line over several physical lines. There are essentially two families of line continuation policies, which we will explain with example of HLASM and COBOL. In HLASM [11, 57, 78], which is the second generation language used on IBM mainframes since System/360, the continuation marker is set on the *continued* line: adding any nonspace character on column 72 will cause the line directly following it, to be concatenated into it. The inconvenience of having only one fixed place to initiate a line break, is alleviated by the option of duplicating the splitting symbol at column

72 leftwards, in which case the entire group is ignored. To illustrate:

```
          ...                long strin*
g constant"
```

is exactly the same as

```
          ...                long ******
string constant"
```

We note that this (sub)feature is not just a pleasant addition/complication, but a necessity: since all leading spaces are stripped from the continuing line, there is no alternative way to split lines on a space.

COBOL handles line continuations differently [61]. The continued line is unaware of it being continued. Instead, the next line carries a continuation marker that turns it into a continuing line, and its contents are appended to the previous line's contents. The marker is usually a dash in column 7. There are many subclauses and rules, for instance preventing line splitting in the middle of certain classes of tokens (e.g., == may not be split) and imposing notational restrictions (e.g., if the split happens in the middle of a string literal, the continuing string needs to start with a double quote to signal explicitly to the parser how many of its leading characters must be skipped). In this case such additional rules are not necessary: they allow marginally more efficient processing of tokens within the original compiler, and a slightly nicer formatting of split strings — the core expressiveness stays the same, so we do not include them in BabyCobol.

There are two ways in which the latter of the presented policies is more suitable for BabyCobol. First, placing the marker on continuing lines is *less* natural in language processing, since it requires lookahead or a comparable infrastructure, since we need to peek into the next line in order to decide on an appropriate action for the current one. Second, it is easier to distil the core functionality from it, stripped from all auxiliary features, which is one of the design goals of BabyCobol.

3.3 Case Insensitivity

Existing programming languages can be roughly classified into three groups: those that distinguish identifiers that only differ by their capitalisation; those that treat variant capitalisations of the same name as being identical; and those that attach semantic meaning to capitalisation. C, C++, Java, C# and many others fall into the first category, and there variables `foobar`, `FooBar` and `FOOBAR` can coexist within the same context. Pascal [75], Forth [51], BASIC [37], COBOL [61, 62], SQL and many 4GLs are case insensitive and thus treat `foobar`, `FooBar` and `FOOBAR` as just different spellings of the same identifier name. Prolog [73], Haskell [30], Rascal [39] and Go [53] are examples of languages where capitalisation has semantic meaning. For example, in Prolog variable names must start with an uppercase letter, and functor names with a lowercase letter.

Since case insensitivity is still rather common among legacy languages, and most modern parser generators do not support it naturally out of the box, we decided to make BabyCobol case **in**sensitive. However, the next subsection introduces a little twist to this aspect, preventing simplistic solutions just indiscriminatingly uppercasing or lowercasing the entire input with the exception of string literals.

3.4 Keywords ≠ Reserved Words

An internationally popular anecdote shared among developers in any languages, concerns variations of the following PL/I code:

```
IF THEN=ELSE THEN ELSE=IF END;
```

This is a piece of perfectly working PL/I code since its keywords are not reserved. In practice there is some variation among PL/I compilers, and some fail to correctly parse and compile programs that use keywords as variable names too liberally, but officially this is what is expected and stated explicitly in the language documentation [60].

We adopt this language feature for BabyCobol, with one little adjustment: we demand for BabyCobol keywords to be uppercased, so that ambiguities only concern identifier names that are also written in upper case. This differentiation has been seen in some lesser known DSLs and 4GLs, but here it fits especially well because it demands the compiler engineers to treat the case insensitivity and tokenising in general, very seriously and develop a detailed solution even if it is painful.

3.5 Whitespace Insignificance

Just like PL/I is known for its non-reserved words example, the following two pieces of FORTRAN code are similarly well-known even to non-FORTRAN developers:

```
DOI=1,10
```

is a looping construct that runs the subsequent statement 10 times for various values of the loop variable `I`, while

```
DOI=1.10
```

is a simple assignment to a variable called `DOI`. A little lexical difference between a dot and a comma leads to a great difference in their syntactic and semantic interpretation, for which the parsing step must account. The core problem behind this is that FORTRAN compilers have complete disregard for whitespace: spaces and tabs were simply ignored if encountered in the input program. From the language designers' point of view this could lead to more readable programs since programmers could be writing `ANYWHERE ON EARTH = UTC - 12` without inventing workarounds like camelcasing in Java-like languages, underscores in C-like ones or dashes in COBOL-like ones. In practice this feature was rarely used for good reasons, but provided endless potential for ambiguities and mistakes, and in later languages was abandoned by sacrificing this "natural" style of variable naming. Interestingly, early dialects of BNF and EBNF also allowed spaces

in nonterminal names, and separated adjacent nonterminals in a sequence with a space. By now almost no grammar definition formalism supports it, everyone switched to more concise notations where the space serves as a separator in a sequence.

Again, to prevent simplistic solutions, we introduce one exception to the rule, and demand spaces around the dash/minus symbol when it is used as an infix arithmetic operator. This is an actual rule existing in COBOL [61], which allows variable names like `ACCOUNT-NUMBER` and in later versions also has natural infix expressions for statements like `COMPUTE` or `EVALUATE`. We intentionally do not include `COMPUTE` since it conveniently unifies `ADD`, `SUBTRACT`, `MULTIPLY` and `DIVIDE` statements, and its lack has a good chance of tempting potential compiler engineers into writing duplicated code for them. Thus, we decide to allow infix expressions in `EVALUATE` expressions and inside conditions of `IF`s. For instance, one could write `IF ACCOUNT-NUMBER = 0 THEN NEXT SENTENCE` which would mean a comparison of the current value of one variable `ACCOUNT-NUMBER` to zero, or `IF ACCOUNT - NUMBER = 0 THEN NEXT SENTENCE`, which would do the same to a difference between values of two variables: `ACCOUNT` and `NUMBER`.

3.6 Lexical Imports

The last lexical feature we will be adding to BabyCobol is inclusion of one program into another. In COBOL this is done by means of a `COPY` directive which literally expands into contents of the included file. If the result yields a compile error, it is reported at the correct location and file, but more complex relations between the included file and the including program are not investigated. Since the inclusion is done verbatim anyway, there is also an extra clause called `REPLACING` which takes two strings and a filename, and replaces all occurrences of one string with the other, during expansion. There are two natural ways to use COBOL's `COPY REPLACING` directive, both used relatively often in legacy code. First, one can prepare special files that are not compilable on their own, and use `COPY` to replace their uncompilable parts with new compilable ones, from different places of the same system. This way, there is some guarantee that such files will only be used through `COPY` expansions. The second way is brute force code reuse: for instance, data structures residing in a separate files because they are used by two programs, can be included into another program which changes the original name prefix. Operating with such files is very error-prone but exemplifies daily troubles of a mainframe programmer. Dealing with all these possible errors and reporting them faithfully and precisely, exemplifies daily troubles of a legacy language processor developer.

Technically this directive is a workaround because some COBOL compilers (most notably the original IBM COBOL compiler for mainframes) support `REPLACE BY ··· REPLACE OFF` directives which can be used to transform any program code on the fly before compiling it, but they do not support `COPY` directives when replacement is on. However, it is problematically enough for compiler constructors to support such replacements at preprocessing time, and putting a general framework for them does not seem to provide any additional conceptual challenges. Thus, in the spirit of keeping BabyCobol minimal, we include one pseudostatement that expands to the contents of the included file, modulo explicitly specified verbatim replacements.

3.7 Program Composition

In COBOL, a program consists of several prescribed *divisions*. Each division has a number of *sections* typical only for that particular division. The `PROCEDURE DIVISION` consists of arbitrarily named sections which contain *paragraphs*, both sections and paragraphs being both named and callable. A paragraph consists of *sentences*, which in turn consist of *statements*. This additional level of structural units between a callable unit (similar to a procedure, a function or a method in other languages) and a statement of which there can be only several kinds well-known in advance, is typical for some legacy languages and delivers a number of complications for compiler writers as well as for software reverse engineers.

Normally there are four top-level divisions in a COBOL program: the *identification* division containing simple paragraphs for setting `PROGRAM-ID` as well as optionally the `AUTHOR` of the program; the *environment* division used for specifying many details about the input/output system, and setting some global flags; the *data* division for describing user-defined types; and the *procedure* division for executable code. For BabyCobol, we keep the identification division in its minimal form, ignore the environment division, narrow the data division to two basic data types, and keep the procedure division practically the same as it is in COBOL.

Statements in Java must be separated by a semicolon. Statements in COBOL do not have to be separated from one another, but they *can* be separated with the help of a period. Such sequences of statements between one period and the next one, are indeed called sentences. There is a special construct `NEXT SENTENCE` [62] which redirects the execution to the point right after the next of the current sentence. In COBOL it is a special subclause of the conditional statement, but for the sake of avoiding accidental complexity we will make it an actual statement in BabyCobol.

4 Semantics of Expressions

4.1 Declarations and Data Types

In COBOL and most COBOL-inspired 4GLs the distinction between a variable and its data type is very thin. More common than not, the composite type of a variable is defined together with its declaration, and is only used once. Data in a type is organised in levels which are explicitly specified (the numbers do not mean much to the compiler, as long as

they increase when we descend deeper and increase when we return — they are crutches replacing proper nesting hierarchy). Abstracting from most details (`DATA DIVISION` takes up 100 pages out of 700 of COBOL's newest *Language Reference* [61, pp.161–260]), we focus on three major features: picture clauses, occurs clauses and like clauses.

Picture clauses are absolutely dominant and omnipresent in COBOL code, relatively widely used in PL/I code, and common to 4GLs (besides the inspiration that 4GLs owe to COBOL, it is near-trivial to support COBOL's picture clauses in a language that compiles to COBOL), yet completely absent from almost any other programming language in existence. The necessity of emulating picture clauses in other languages when a language conversion is performed as a part of legacy software migration, is one of the major reasons such conversions regularly fail to deliver and satisfy customers' expectations [67]. A picture clause is a primitive type definition which revolves around a given pattern for storage and displaying value. For instance, if a field `VAR` is defined as `VAR PICTURE IS $999V99`, then five decimal places need to be stored, and each time it is printed, displayed or otherwise exposed, its leftmost symbol is a verbatim dollar sign, followed by three digits, then by a decimal dot (or a comma, depending on the end user's locale and other configuration options), and finally by two more digits. In modern high-level programming languages the closest analogy to a picture clause would be an object field with a getter that formats its contents according to the pattern, and a setter that parses the provided value according to the same pattern. There are many tricks and options in specifying a picture pattern, but for BabyCobol we keep only the following special characters (simplified whenever possible, everything is more complex in COBOL):

- `9` — any digit
- `A` — an alphabetic character or a space
- `X` — any single character
- `Z` — a leading digit, disappearing into space if zero
- `S` — a sign (`+` or `-`, space treated as a plus)
- `V` — a decimal separator (usually `.` or `,`)

Any other characters are accepted at their verbatim value. Any number of consequent characters of the same value can be replaced with `C(N)` where `C` is the character and `N` is the number of time it is repeated. Hence, for instance, `999999V99` is the same as `9(6)V9(2)`, and `X(20)` is just a declaration of any string value up to 20 characters long.

An `OCCURS` clause essentially allows to define arrays:

```
01 ARRAY OCCURS 20.
   03 ELEMENT PICTURE IS 9(15)V99.
```

After such a declaration, `ELEMENT` becomes ambiguous and must be used as `ELEMENT OF ARRAY(1)` up till and including `ELEMENT OF ARRAY(20)`.

Finally, like clauses take the following structure:

```
03 NEW-FIELD LIKE ANOTHER-FIELD.
```

This allows for `NEW-FIELD` to reuse the type that was defined when `ANOTHER-FIELD` was defined. `LIKE` clauses are absent from COBOL, but quite prominent in PL/I [60], RPG [59], REXX [19] and many 4GLs, so we include them in BabyCobol. Notably, since `OCCURS` is an additional clause, it is not treated as an essential part of the type by 3GL and 4GL compilers, so using a `LIKE` on an `OCCURS` type will declare a variable of a base scalar ("non-occurring") type, unless the clauses are combined: `X LIKE Y OCCURS 42`.

With respect to semantics, it is important to remember that all data types are decimal position based, not bit based, so overflows will also happen not at, say, 65535 or 32767, but at 99999. All overflows are silent.

Arithmetic operations (`SUBTRACT`, `DIVIDE`, `MULTIPLY` and `ADD`) may only be performed on atomic fields which pictures are `A/X`-free (i.e., not on arbitrary strings, not on arrays, not on composite data structures). Infix operators that we allow as first arguments of `IF` and `EVALUATE` statements (`+`, `-`, `*`, `/` and `**`) have the same limitation, except the first one (`+`) which is also defined on arbitrary strings. Comparison operators work symbol per symbol, space-padding the shortest of the values on the left. All these rules have been inherited by BabyCobol from COBOL for simplicity reasons.

4.2 Sufficient Qualification

Qualified variable names in COBOL [61], PL/I and most 4GLs grow long and tiresome for developers to write fully. Instead, such legacy languages employ a scheme called sufficient qualification, where only a part of the path towards the field needs to be communicated to the compiler, the rest is inferred automatically. For instance, if we have a data structure `FOO` which has a child data structure `BAR` which has a field `F`, then `F OF BAR OF FOO` would be its full qualification. In the context of a program that does not have any competing `F`s anywhere, just saying `F` may be enough. In other cases, the programmer will write `F OF FOO` to indicate the beginning and the end of the path, leaving the middle part for the inference, or `F OF BAR`, leaving the compiler to find the root. Whenever insufficient qualification causes an ambiguity, the compiler tries to resolve it from the type information and the context (for example, in the context of `F>0` only expanded qualifications with numeric picture patterns will be considered). If all resolution strategies fail, the compilation fails with an appropriate error message. Insufficient indices to occurring fields are assumed to be 1s.

4.3 Figurative Constants

COBOL, as well as some other legacy languages, employ so called "figurative" constants next to normal fixed literals. A figurative constant is fixed for any given context, but not fixed in general. For example, `MOVE SPACES TO X` will assign twenty spaces to a `X(20)` typed field, three spaces to a `Z(3)` representing a decimal zero, or even go deeper and populate all lower levels of data similarly with spaces

if X is a compound data structure. Besides SPACES, we keep LOW-VALUES and HIGH-VALUES and for the sake of simplicity ignore other figurative constants found in COBOL, such as ZERO, QUOTE, NULL and ALL [62, p.28]. As can be intuitively understood by their names, LOW-VALUES and HIGH-VALUES represent the minimum and the maximum allowable values for a given type. For example, MOVE LOW-VALUES TO X will assign it a zero of X PICTURE IS 99 but a value of −99 instead if X PICTURE IS S99.

4.4 Default Values

There are no compiler errors in REXX that complain about unknown variables. If an undeclared variable is used, it is considered thereby declared and instantiated with its own uppercased name (so the default value of ACCOUNT-NAME would be "ACCOUNT-NAME"). We include the same feature in BabyCobol. The inferred type of such an auto-declared variable PICTURE IS X(N), where N is the length of its name. Coincidentally this also prevents developers from accidentally turning their dash-containing variable names into infix subtraction expression in many cases, since their default values are strings, and subtraction is not defined on strings. Yet again, it does shift the burden to the compiler developers to incorporate a symbol table lookup into the error reporting facility to decide whether to report about incorrect whitespace or about attempted string subtraction, because they mean vastly different things to the end programmer.

4.5 Contractions in Conditions

COBOL has a particularly challenging feature that looks simple at first glance. Instead of repeating themselves when writing expressions, developers are allowed to contract them and write IF X = 2 OR 3 when they mean IF X = 2 OR X = 3. Contractions can start at any position, so expressions like IF X = 2 OR 3 OR > 10 are also allowed and processed correctly. At the same time, operator priorities work normally and thus can bind full expressions to contracted expressions that belong to a different group by contracting. Thus, IF X = 42 OR 3 AND Y < X is the same as IF X = 42 OR (X = 3 AND Y < X). This behaviour may seem counter-intuitive but has been confirmed by running test cases with exhaustive coverage on a mainframe. Parsing such contracted expressions correctly poses a considerable challenge since the subexpressions need to both be available in linear succession in case the next one will need expansion based on its predecessors, and at the same time must get gathered into a tree structure following the rules of Boolean and decimal arithmetics. This is well within any compiler constructor's capabilities, but is rarely demanded by non-legacy languages.

5 Semantics of Statements

5.1 Unconditional Transfer of Control

GO TO has been declared worthy to be considered harmful since 1968 [20]. Even though there was some discussion on the universal applicability of this judgement and the universal need to replace all unconditional GO TOs with structural equivalents [40, 56], claims that they should be used as the main way of expressing control flows, were considered outdated already in late 1980s [36]. However, for a compiler writer the presence of GO TOs is problematic only if certain formalisations are used (for instance, big-step operational semantics suffers from it and requires a relatively ugly rewrite of an otherwise straightforward system of beautiful formulae). At the code generation phase, GO TO remains relatively painless. Unfortunately, there is another level of complications, commonly encountered in legacy software and collectively overlooked in papers about harmfulness of GO TO. Consider the following piece of COBOL code:

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

(To explain the possibly obvious: we declare a new paragraph and populate it with a single statement of unconditional transfer of control).

Then, after the following statement is executed from any part of the same program:

```
    ALTER EXIT-ON-ERROR TO PROCEED
    TO EXIT-ROLLBACK-RECORD.
```

it makes it so that the original piece of code acts as if it had been written as follows:

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

In other words, the ALTER statement redirects an existing GO TO statement to go to a different place than originally intended. Self-modification is not entirely uncommon in high-level languages: it is relatively omnipresent in prototype-based object-oriented programming languages like JavaScript, and all homoiconic languages like LISP [50], Clojure [31], Rebol [28], Scheme [66] or Racket [22] allow free treatment of anything as either data (with decomposition, mutation, etc) or code (with executability), since their homoiconicity by definition implies the "code as data" paradigm. However, this particular combination with a low-level construct like an unconditional GO TO with an ability to modify it, is endemic to legacy languages. Hence, it must be included in BabyCobol.

5.2 Structured Control Statements

COBOL's PERFORM covers everything all modern programming languages' looping constructs like for, do and while can do, and much more, including calling a procedure within the same program (since originally it was required for a looping construct to give a label or an address of its target instead of the actual statement, like for DO in FORTRAN).

Concentrating such number of concepts and subclauses in one statement is not sensible even for a language that is intended as a challenge: different variants of the same construct will just get near-trivially desugared into disjoint intermediate representations. Hence, we split COBOL's `PERFORM` into two BabyCobol statements, roughly corresponding to a simplified version of the in-line perform and the out-of-line perform. `PERFORM` in BabyCobol acts like an out-of-line perform in COBOL [62], calling a procedure (a section or a paragraph) and returning to continue execution on the next statement. It supports a particularly problematic way to call several subsequently written paragraphs as one, which is known to cause maintenance difficulties for obvious reasons. `LOOP` in BabyCobol acts like an in-line perform with one extra complication borrowed from AppBuilder and other 4GLs: its clauses (like `UNTIL` or `WHILE`) do not have to occur at the beginning or the end of the loop, but instead anywhere within its body.

5.3 Name-Driven Assignment

Assignments in modern languages are mostly straightforward, occasionally imposing complex ownership rules as in Rust [48]. Given the once-off type system of COBOL (§ 4.1), it is not surprising to find a name-driven deep overlay assignment there. `MOVE CORRESPONDING A TO B` [62] is a COBOL statement that goes through all fields of `A` and maps the values of all name-matching fields with `B` to their counterparts, discarding the rest. In modern languages this is only possible through reflection, but there are entire 4GLs like AppBuilder that are built around the idea of making this kind of assignment efficiently compilable. We define BabyCobol's `MOVE` to do what AppBuilder's `MAP` or COBOL's `MOVE CORRESPONDING` does.

5.4 Exception Handling

REXX's `SIGNAL` statement [19] and PL/I's `ON` units [60] inspired us to include error handling into BabyCobol. To keep it minimal, we only allow one handler for all kinds of runtime errors. If an exception happens (say, an index goes out of bounds), the handler is called, after which the execution continues normally. In that sense, it is closer to algebraic effects than to Java-style exceptions that can break through several layers in the execution stack.

6 Inspirational Languages Summary

For size considerations, we cannot include a full table comparing each possible BabyCobol statement to its equivalents in languages of different generations (2GL, 3GLs and 4GLs), in the paper. Below we briefly report, per language, which of its constructs are known among industrial legacy engineers to be challenging, how they were considered for inclusion in BabyCobol, and whether they ended up in the language, to document our line of reasoning. In every case claimed equivalence of constructs among languages should be understood in a broad sense. For example, BabyCobol's `PERFORM P X TIMES` directly translates to literally the same statement in COBOL, but it will look like `DO P I=1,X` in FORTRAN (with an introduction of a temporary variable) and like `DO X CALL P END` in REXX (with a combination of two constructs, and a different idea in designer's mind about what it is we are trying to *do* — `X` or `P`). Similarly, BabyCobol's `DISPLAY X` in CLIST is just `WRITE X`, but in FORTRAN it requires a separate `FORMAT` statement labelled, say, `P`, after which can come `PRINT P,X`. In that case the information of what to put into the arguments of that `FORMAT` by a hypothetical BabyCobol-to-FORTRAN compiler, must come from the knowledge of the type system.

HLASM [57] ("High Level ASseMbler"), sometimes called a "*second* generation language" (2GL) to set it apart from raw machine code, is the main assembler language for IBM mainframes (from System/360 up to z15; earlier 1400 series used SPS and Autocoder, and 700/7000 series used FAP and IBMAP instead). It exists since 1964 as the Basic Assembler Language (BAL) and got its current name in 1992 after a series of incremental changes. For many projects where it is used, it was chosen not for performance but for simplicity reasons. We did use HLASM as an example when discussing line continuations in § 3.2, but ended up not using that much of the language itself for BabyCobol, even though it contains almost a thousand instructions and several hundreds macros. In particular, HLASM instruction `EXECUTE` was considered for inclusion: its semantics is to take several bytes from an arbitrary given place in memory, apply a mask to it and execute them as code [78]. Even if it is raised from the low level of assembler, it would mean treating arbitrary data as code — a feature known as "eval" [54] in mainstream languages such as JavaScript, PHP, Perl, Python, Ruby, Lua, Forth, as well as Smalltalk and LISP. It is very problematic for compiler developers, up to the point when program analysis cannot remove all evals from a program being compiled, and its compiled version needs to contain an interpreter for the same language. Hence, full compilation of eval-containing programs is equivalent to solving higher Futamura projections [24] which is a theoretically highly nontrivial and practically commonly unsolvable problem. Our design goal for BabyCobol was to provide challenges for a compiler constructor, but if these challenges are impossible to overcome, it defeats the original purpose.

Mainframe alternatives to HLASM include so-called "third generation languages" (3GLs — traditional high level ones) and "fourth generation languages" (4GLs — domain-specific languages for report processing, database communication, transaction handling, interfaces, model-based code generation, etc). **COBOL** [61] ("COmmon Business-Oriented Language") is a 3GL, one of the oldest high-level computer languages in existence, and remains one of the most popular

ones even 60 years of its inception. COBOL is also by far the most commonly encountered programming language in legacy software, up to the point where even in systems dominated by 4GLs, it is used as the compilation target for 4GLs as well as a common language for manually written glue code. Hence, it was the most logical for it to serve as the main foundation for the design of BabyCobol (as well as the inspiration for its name). COBOL contains 39 statements, out of which 11 ended up in BabyCobol in the most straightforward fashion, with only superficial simplifications. BabyCobol's `MOVE` took after COBOL's `MOVE CORRESPONDING`, a special subclause of the general assignment statement. BabyCobol's `DISPLAY` combines functionality of COBOL's `DISPLAY` and `STRING` statements. COBOL's `PERFORM` was after lengthy considerations considered overly complex and was split into two separate statements in BabyCobol: `LOOP` (usually referred to as "the in-line perform" among COBOL programmers) and `PERFORM` (the "out-of-line" version).

CLIST [58] ("Command List") is an interpreted imperative 3GL that is still in use on occasional mainframe systems, even though it is largely inferior to alternatives: its abstractions are lower level and more leaky compared to REXX, and its speed of execution is much slower than of any compiled language. CLIST consists of 31 different statements, most of which are either way too specific (e.g., `LISTDSI`) to be included in BabyCobol, or duplicating existing COBOL functionality (e.g., `WRITE` and `WRITENR` do the same as `DISPLAY`, `SET` is just a flipped `MOVE` and `EXIT` is a `STOP RUN`). Yet, CLIST contributed to BabyCobol in one significant way: its `GOTO` statement has an option of using a variable as a target instead of a verbatim label. This is more powerful and concise than anything `DEPENDING ON` clause can offer, and thus fits BabyCobol in two ways: by simplicity of expression and by problems caused for compiler construction and program analysis.

REXX [19] ("Restructured Extended Executor") is a scripting and macro 3GL that can be interpreted or compiled, and could be easily explained by labelling it "the mainframe's Perl". It contains some instructions obviously inspired by CLIST (e.g., `SELECT/WHEN/OTHERWISE`), some hiding under different names (it uses `PULL` for CLIST's `READ` or COBOL's `ACCEPT`, and `SAY` instead of `WRITE` or `DISPLAY`, resp). REXX also contains an eval-like instruction `INTERPRET` which is certainly problematic for compiler constructors, but just like HLASM's `EXECUTE` discussed above, it is *too* problematic. Any *compiled* program with evals will have in the worst case to include a complete interpreter of the same language. Our goal for BabyCobol is to make compilation challenging, not impossible. Another powerful instruction of REXX is `PARSE` that can be used to parse standard input pieces, procedure arguments, program metainfo or any arbitrary expression as a sequence of variables and literals. This is equivalent to a pure BNF non-recursive parser with variables as nonterminals and literals as terminals. REXX's `SIGNAL` instruction is used for sophisticated error handling, and since error handling is an interesting and challenging mechanism that can also interfere with some implementations of control statements and unconditional transfer, it was included in BabyCobol. There are 7 different kinds of exceptions that the original `SIGNAL` can handle, including even syntax errors in the program, but in BabyCobol we simplify them to one. The deciding factor in preferring `SIGNAL` to COBOL's and PL/I's `ON` units was simplicity: in REXX it is one instruction that provides one uniform error handler, while others allow for fine-grained error handling per statement. Another feature of REXX that made it into BabyCobol is default variable values. Since in REXX all variables store strings, it naturally allows undeclared variable names to be used as literals, such that `SAY hello` prints the value of the variable `hello` if it is defined and an uppercase `HELLO` otherwise.

FORTRAN [7] ("FORmula TRANslator") is a compiled general-purpose programming language for solving numerical problems. It has many dialects and survived several versions that significantly changed both the appearance of its programs and their treatment. For the purpose of this project, we focused on the oldest of the publicly available FORTRAN manuals for the IBM 704 EDPM. It contains many statements that are way too hard to understand, let alone implement, for a modern developer, and most of those have by now lost their relevance altogether. For instance, it has at least seven output-related statements: `FORMAT`, `PUNCH`, `PRINT`, `WRITE OUTPUT TAPE`, `WRITE TAPE`, `WRITE DRUM` and `END FILE`. Yet, it also has several interesting and challenging instructions that are still relevant to interpreting, compiling and migrating legacy code, and we zoom in here on three of them. First, there is an additional statement `PAUSE` occupying the conceptual space between `CONTINUE` that simply continues execution by doing nothing, and `STOP` which completely terminates the program. `PAUSE` stops the program while preserving its state entirely, so the next click on the START button by the user allows the execution to continue. After some consideration we decided not to include the `PAUSE` statement in BabyCobol, since apparently it is also not used much in code surviving up to this day. Second, FORTRAN contains a `FREQUENCY` statement which is used to provide hints to the compiler on optimizations of the user code, by explicitly communicating estimated frequencies of choosing particular branches at decision points (`IF`s and `DO`s). This feature is too heavyweight for BabyCobol, but it is worth considering in some form for its possible extensions. Finally, let us consider FORTRAN's `GO TO`s closely. There are three forms of `GO TO` in FORTRAN I: the unconditional `GO TO`, the computed `GO TO` and the assigned `GO TO`. The unconditional variant is the standard version that just transfers control to a different location. The computed variant transfers control to an indexed location from a list and is thus a shorthand notation for a sequence of `IF`s with corresponding `GO TO`s. The assigned `GO TO` has such a name because of another

statement ASSIGN which semantics very closely resembles COBOL's ALTER. However, it is written this way [7, p.17]:

```
ASSIGN 12 TO N
GO TO N, (7, 12, 19)
```

On one hand, the ASSIGN command is more than a simple value assignment (which could have taken the form of "N = 12"), since it signals to the compiler that the variable N will be used as a label name and from that point on cannot be used in arithmetic expressions. On the other hand, the GO TO command, which can be very far removed from the actual target assignment in real code, contains a list of allowed assignable line numbers, and thus helps the programmer to read and understand this code, as well as allows the compiler to statically check all related ASSIGNs against this list. This subtle point demonstrates the difference between making the life of a developer difficult (which is definitely the case with COBOL's GO TO and ALTER since at the point of change nothing indicates possible range of altered values) and making the life of a compiler developer difficult (by adding more constraints imposed on seemingly unstructured constructs in a way that allows compile-time static analysis and error reporting). Our goal is the latter.

PL/I [60] ("Programming Language One") is one of the most complex programming languages in all aspects: the number of nonterminals in PL/I and COBOL grammars written in the same notation, is comparable, but the complexity is considerably higher (taken as the length of production rules and the branching of them at ? and ∗-combinators). The sheer number of terminal symbols is so high that keywords and built-in function names in PL/I are not reserved words so that developers still have a chance to call their variables "MODE", "NOTE", "GET" or "PUT". The semantics of each statement is often also difficult to grasp and contains subclauses that behave differently. It is safe to assume that PL/I covers everything. It contains error handling similar to REXX but much more complex: it covers 24 different kinds of them (REXX has 7, BabyCobol has 1), and they can also be fine-tuned per kind and per statement (with "condition prefixes"), specifying exactly what the behaviour should be, whether the debug information should be printed, whether the implicit system action needs to be taken, which procedure must handle it, and even during the handling itself the execution can be stopped, reverted, restarted, etc. There are 4 explicitly differently specified variants of PL/I's DO loop, some having the complexity of COBOL's "in-line PERFORM" with all its ad hoc clauses, while others almost reaching C-style low-level clarity by communicating which expression exactly to reevaluate on each iteration (i.e., the REPEAT clause in DO type 3). PL/I also has built-in facilities for writing multi-threaded code, which is a problem in almost all other 3GLs and 4GLs.

4GLs ("Fourth Generation Languages") were highly praised in the 1960s through 1980s [47, 63] for being non-procedural high level specification languages that allow software developers to write concise yet readable code that is easier to design, develop, evolve and maintain. Their implementers were mostly focusing on the conciseness of the code that had to be written, often overlooking other DSL design principles that are valued today (developer efficiency, learnability, tool support, debuggability, etc). In the modern world such languages are in decline and are commonly seen as legacy [21], and owners of codebases and portfolios largely relying on such 4GLs, actively undertake steps towards their retirement, investing millions in multi-year plans for software modernisation [38, 67]. Among 4GLs we have looked into CA's **COOL:GEN** [15], **IDEAL** [14] and **VISION:BUILDER** [13], Information Builders' **FOCUS** [34], Magic Software's **AppBuilder** [45], Software AG's **NATURAL** [64], jBASE's **jBASIC** [35], and IBM's **Informix** [25] and **PACBASE** [32]. Quite commonly only superficial information is available for researchers who are not current paying customers of the language' compiler. Sometimes there was barely enough public information available at the official websites to examine the list of statements but not their precise nor detailed semantics. The design of many of these proprietary 4GLs was heavily inspired by both COBOL and PL/I. For example, AppBuilder code mostly relies on the MAP statement which is identical to BabyCobol's MOVE, but also uses abbreviated PL/I names like DCL for declarations and PROC for procedures. Its DO loop also contains FROM, TO, INDEX, BY (but no UPTHRU and DOWNTHRU) clauses, as well as the WHILE clause which is detachable (it can occur anywhere within the body of the loop, not just at the beginning or the end). Yet it still does not reach the impressive complexity of PL/I's DO which essentially mixes four different looping statements into one, each having quite a number of subclauses [60, pp.212–223]. We have incorporated this detachable subclause of AppBuilder's DO/WHILE into BabyCobol's LOOP.

7 Conclusion

A simplified version of an EBNF definition of BabyCobol is provided on Listing 2. The main website supporting BabyCobol, can be found at **https://slebok.github.io/baby**, it contains a more complete and growing language reference with detailed directions, explanations, code examples, etc. This project is a part of a bigger initiative called SLEBoK, for Software Language Engineering Body of Knowledge, a community-wide effort to provide a unique and comprehensive description of the concepts, tools and methods developed by the SLE community. It features artefacts, definitions, methods, techniques, best practices, open challenges, case studies, teaching material, and other components that would help students, researchers, teachers, and practitioners to learn from, to better leverage, to better contribute to, and to better disseminate the intellectual contributions and practical tools

```
Sentence          ::= Statement+ .
Statement         ::= ACCEPT Identifier+
                    | ADD Atomic+ TO Atomic (GIVING Identifier)?
                    | ALTER ProcedureName TO PROCEED TO ProcedureName
                    | COPY FileName (REPLACING (Literal BY Literal)+)?
                    | DISPLAY DisplayExpression* (WITH NO ADVANCING)?
                    | DIVIDE Atomic INTO Atomic+ (GIVING Identifier)? (REMAINDER Identifier)?
                    | EVALUATE AnyExpression WhenBlock* END
                    | GO TO ProcedureName (OR ProcedureName)*
                    | IF BooleanExpression THEN Statement+ (ELSE Statement+)? END?
                    | LOOP LoopStatement* END
                    | MOVE MoveExpression TO Identifier+
                    | MULTIPLY Atomic BY Atomic+ (GIVING Identifier)?
                    | NEXT SENTENCE
                    | PERFORM ProcedureName (THROUGH ProcedureName)? (Atomic TIMES)?
                    | SIGNAL (OFF | ProcedureName) ON ERROR
                    | STOP
                    | SUBTRACT Atomic+ FROM Atomic (GIVING Identifier)?
DisplayExpression ::= Atomic (DELIMITED BY (SPACE | SIZE))?
MoveExpression    ::= Atomic
                    | HIGH-VALUES
                    | LOW-VALUES
                    | SPACES
WhenBlock         ::= WHEN Atomic+
                    | WHEN OTHER
AnyExpression     ::= ArithmeticExpression
                    | StringExpression
                    | BooleanExpression
ArithmeticExpression ::= Atomic
                    | ArithmeticExpression ArithmeticOp ArithmeticExpression
StringExpression  ::= Atomic
                    | StringExpression + StringExpression
BooleanExpression ::= TRUE
                    | FALSE
                    | ArithmeticExpression ComparisonOp ArithmeticExpression
                    | NOT BooleanExpression
                    | BooleanExpression BooleanOp BooleanExpression
LoopStatement     ::= VARYING Identifier? (FROM Atomic)? (TO Atomic)? (BY Atomic)?
                    | WHILE BooleanExpression
                    | UNTIL BooleanExpression
                    | Statement
ComparisonOp      ::= = | > | < | >= | <=
BooleanOp         ::= OR | AND | XOR
ArithmeticOp      ::= + | - | * | / | **
Atomic            ::= Literal | Identifier
ProcedureName     ::= SectionName | ParagraphName | Identifier
Identifier        ::= Name | Name ( Index )
```

Listing 2. Syntax of BabyCobol statements and expressions. We abstract from some understandable details like optional bracketing of expressions, as well as from complex features discussed in § 3, § 4.2 and § 4.5. There are also many static semantic details not expressed here: `ADD` and `SUBTRACT` are only allowed their second argument to be literals when the `GIVING` clause is present; `DIVIDE` and `MULTIPLY` may have only one second argument when the `GIVING` clause is present; end of a sentence terminates all currently open `IF`s without any need of matching `END`s, etc.

and techniques coming from the SLE field. The online language manual is written traditionally top-down, and goes through each of the statement kinds one by one, explaining them in detail. Contrary to that style, in this paper we reported on BabyCobol from two sides: first by explaining its features by examples of known problematic constructs, giving examples from legacy languages (§ 3 – § 5); and then again by going through the list of popular legacy languages reflecting on which parts of them were considered for inclusion and which ended up as BabyCobol constructs (§ 6).

By choosing BabyCobol to be a separate language and not a strict subset of COBOL, we unfortunately forfeit the way MiniJava and similar projects deal with specifying the language semantics: by stating that the semantics of the new language is equivalent to the semantics of literally the same piece of code in the original language. There are mainly two reasons for that. First, Java is a great language to make subsets of, since it has one reference implementation which is freely available, so even if the documentation is unclear, one can always write a test case and compile it. This is not the case with COBOL: it has hundreds of dialects with different semantics, and most compilers are inaccessible to the general public anyway, and/or target platforms such as IBM mainframes, which also require hard to arrange expensive access. Second, it has been made very clear to us by several industrial compiler developers that problems of implementing COBOL alone do not do justice at representing all kinds of possible problems of implementing tools for 2GLs through 4GLs. In particular, even fellow 3GLs like REXX, CLIST and PL/I come with their own sets of implementation problems that are not necessarily related to the set of COBOL implementation problems at all. Hence, we opted for a superset of a union taken from a collection of problems of COBOL, as well as all kinds of other legacy languages.

As one of the possible implications, we may have gained undefined or undesirable features of BabyCobol in places where semantics of one of its features inherited from one legacy language, interacts with another feature inherited from a different legacy language. The feature interactions we have observed so far — for example, how `PERFORM THROUGH` behaviour collaborates with the `GO TO` behaviour jumping within the range of paragraphs being performed or outside of it — were verified as real issues, causing the same problems in BabyCobol and COBOL, and almost verbatim portable.

We claim that implementing its parser witnessing case independence, its syntax highlighter witnessing whitespace indifference and keywords not being reserved, its code completion for sufficiently expanding qualified names, its code generator for several vastly different control structures, and other forms of compilers and compiler-like language processing tools, will help researchers to relate to problems of legacy software handling and train on soothing the BabyCobol before taking on its full grown counterparts. In the meantime we have started using BabyCobol in teaching graduate students the dark hard side of software evolution and maintenance, reinforcing the effect with guest lectures given by practitioners. We are planning to report on our experiences separately, once enough evidence is gathered. Ideally, we would like to eventually run empirical evaluations investigating how non-BabyCobol taught junior developers solve legacy language processing tasks compared to BabyCobol-taught ones, but the practical feasibility of such a study remains to be determined.

Acknowledgements

The author would like to thank Arthur Michener Peters from the University of Texas at Austin, who has suggested the idea of creating such a language at a lunch discussion at SPLASH'19 in Athens on 21 October 2019. Feedback from attendees of the author's keynote speech at BENEVOL'19 on 28 November 2019 and the author's invited talk at PRiML'20 on 6 July 2020, as well as Master of Science students of Universities of Amsterdam (The Netherlands) and Mons (Belgium), who attended guest lectures of the author on 9 December 2019 and 4 March 2020, respectively, and all the anonymous reviewers, was instrumental in leading this paper to its final form. Special thanks go to Bernd Fischer and other members of the IFIP Working Group 2.11 on Program Generation, who attended a presentation on an earlier version of BabyCobol on 17 February 2020, provided valuable feedback and produced the first BabyCobol-based tool that generated test programs from its grammar.

References

[1] Hal Abelson, Nat Goodman, and Lee Rudolph. 1974. *LOGO Manual.* Technical Report. MIT. http://hdl.handle.net/1721.1/6226.

[2] Davide Ancona and Elena Zucca. 2012. Corecursive Featherweight Java. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP)*, Wei-Ngan Chin and Aquinas Hobor (Eds.). ACM, 3–10. https://doi.org/10.1145/2318202.2318205

[3] Dana Angluin. 1980. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences* 21, 1 (1980), 46–62. https://doi.org/10.1016/0022-0000(80)90041-0

[4] Sven Apel, Christian Kästner, and Christian Lengauer. 2008. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proceedings of the Seventh International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 101–112. https://doi.org/10.1145/1449913.1449931

[5] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java: Second Edition.* Cambridge University Press.

[6] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (LNCS, Vol. 3603)*, Joe Hurd and Thomas F. Melham (Eds.). Springer, 50–65. https://doi.org/10.1007/11541868_4

[7] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, and I. Ziller. 1956. *The Fortran Automatic Coding System for*

the IBM 704 EDPM. Programmer's Reference Manual. Applied Science Division and Programming Research Dept., IBM Corporation.

[8] Lorenzo Bettini, Sara Capecchi, and Elena Giachino. 2008. Featherweight Wrap Java: wrapping objects and methods. *Journal of Object Technology* 7, 2 (2008), 5–29. https://doi.org/10.5381/jot.2008.7.2.a1

[9] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. 2002. *MJ: An imperative core calculus for Java and Java with effects.* Technical Report 563. Computer Laboratory, University of Cambridge.

[10] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[11] Volodymyr Blagodarov, Yves Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 221–225. https://doi.org/10.1145/2997364.2997387

[12] Timofey Bryksin, Victor Petukhov, Kirill Smirenko, and Nikita Povarov. 2018. Detecting Anomalies in Kotlin code. In *Companion Proceedings for the ISSTA/ECOOP Workshops*, Julian Dolby, William G. J. Halfond, and Ashish Mishra (Eds.). ACM, 10–12. https://doi.org/10.1145/3236454.3236457

[13] CA Technologies. 2005. Advantage™ VISION:Builder® Advantage™ VISION:Two™ for z/OS Reference Guide r15. B02630-1E, https://ftpdocs.broadcom.com/cadocs/0/b026301e.pdf.

[14] CA Technologies. 2015. CA Ideal™ for CA Datacom® Programming Guide V. 14.02. https://ftpdocs.broadcom.com/cadocs/0/CAIdeal140-ENU/Bookshelf_Files/PDF/ID1402_Programming_ENU.pdf.

[15] CA Technologies. 2016. 5 Ways DevOps Practices Boost Innovation on the Mainframe. CS 200-227965, https://docs.broadcom.com/doc/5-ways-devops-practices-boost-innovation-on-the-mainframe.

[16] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17, 4 (Dec. 1985), 471–523. https://doi.org/10.1145/6041.6042

[17] William R. Cook. 1991. Object-Oriented Programming versus Abstract Data Types. In *Foundations of Object-Oriented Languages (REX Workshop 1990)*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg (Eds.). Springer, 151–178. https://doi.org/10.1007/BFb0019443

[18] William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Shail Arora and Gary T. Leavens (Eds.). ACM, 557–572. https://doi.org/10.1145/1640089.1640133

[19] Michael Cowlishaw. 1990. *The Rexx Language: A Practical Approach to Programming.* Prentice Hall.

[20] Edsger W. Dijkstra. 1968. Go To Statement Considered Harmful. *Commun. ACM* 11 (1968), 147–148. https://doi.org/10.1145/362929.362947

[21] Michael Feathers. 2004. *Working Effectively with Legacy Code.* Prentice-Hall.

[22] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. DrScheme: A Pedagogic Programming Environment for Scheme. In *Proceedings of the Ninth International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP)) (LNCS, Vol. 1292)*, Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen (Eds.). Springer, 369–388. https://doi.org/10.1007/BFb0033856

[23] J. Nathan Foster and Dimitrios Vytiniotis. 2006. A Theory of Featherweight Java in Isabelle/HOL. *Archive of Formal Proofs* (March 2006). https://www.isa-afp.org/entries/FeatherweightJava.shtml

[24] Yoshihiko Futamura. 1982. Partial Computation of Programs. In *Proceedings of the RIMS Symposium on Software Science and Engineering*, Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa (Eds.), Vol. 147. Springer, 1–35. https://doi.org/10.1007/3-540-11980-9_13

[25] G2 Crowd. 2018. Informix® software: An Embedded Database for the Edge and Beyond. https://www.ibm.com/downloads/cas/NQQNEG7K.

[26] Leo Geurts, Lambert G. L. T. Meertens, and Steven Pemberton. 1990. *ABC programmer's handbook.* Prentice Hall.

[27] P. Giles. 1969. Mini-COBOL. *Comput. J.* 12 (Aug. 1969), 208–214. Issue 3. https://doi.org/10.1093/comjnl/12.3.208

[28] E. Goldman and J. Blanton. 2000. *REBOL Official Guide.* McGraw-Hill.

[29] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. 2005. *The Java Language Specification* (third ed.). Addison-Wesley.

[30] Kevin Hammond and Stephen Blott. 1989. Implementing Haskell Type Classes. In *Proceedings of the Glasgow Workshop on Functional Programming (Workshops in Computing)*, Kei Davis and John Hughes (Eds.). Springer, 266–286. https://doi.org/10.1007/978-1-4471-3166-3_18

[31] R. Hickey. 2008. The Clojure programming language. In *DLS*, Johan Brichau (Ed.). ACM, 1. https://doi.org/10.1145/1408681.1408682

[32] IBM. 1998. VisualAge Pacbase 2.5. Pacbase Access Facility Reference Manual. DDPAF000251A, ftp://public.dhe.ibm.com/software/vapacbase/pdf_e/paf251a.pdf.

[33] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[34] Information Builders. 2018. FOCUS for Mainframe and Distributed Systems Technical Library. http://ecl.informationbuilders.com/focus/index.jsp.

[35] jBASE. 2020. jBASE Support Documentation: jBase Basic (JBC). https://jbase.helpjuice.com/36868-jbase-basic.

[36] Dan Jonsson. 1989. Next: the Elimination of Goto-Patches? *ACM SIGPLAN Notices* 24, 3 (1989), 85–92. https://doi.org/10.1145/66083.66091

[37] John G. Kemeny and Thomas E. Kurtz. 1964. *BASIC: A Manual for BASIC, the Elementary Algebraic Language Designed for Use with the Dartmouth Time Sharing System.* Technical Report. Dartmouth College.

[38] Ravi Khadka, Amir Saeidi, Slinger Jansen, Jurriaan Hage, and Geer P. Haas. 2013. Migrating a Large Scale Legacy Application to SOA: Challenges and Lessons Learned. In *Proceedings of the 20th Working Conference on Reverse Engineering*, Ralf Lämmel, Rocco Oliveto, and Romain Robbes (Eds.). IEEE, 425–432. https://doi.org/10.1109/WCRE.2013.6671318

[39] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE CS, 168–177. https://doi.org/10.1109/SCAM.2009.28

[40] Donald E. Knuth. 1974. Structured Programming with Go To Statements. *Comput. Surveys* 6, 4 (1974), 261–301.

[41] Ralf Lämmel and Chris Verhoef. 2001. Cracking the 500-Language Problem. *IEEE Software* 18, 6 (Nov./Dec. 2001), 78–88. https://doi.org/10.1109/52.965809

[42] Christopher League, Zhong Shao, and Valery Trifonov. 2002. Type-preserving compilation of Featherweight Java. *ACM Trans. Program. Lang. Syst.* 24, 2 (2002), 112–152. https://doi.org/10.1145/514952.514954

[43] Manuel Leduc, Thomas Degueule, Eric Van Wyk, and Benoît Combemale. 2020. The Software Language Extension Problem. *Software and Systems Modeling* 19, 2 (2020), 263–267. https://doi.org/10.1007/s10270-

019-00772-7

[44] Luigi Liquori and Arnaud Spiwack. 2008. FeatherTrait: A modest extension of Featherweight Java. *ACM Trans. Program. Lang. Syst.* 30, 2 (2008), 11:1–11:32. https://doi.org/10.1145/1330017.1330022

[45] Magic Software Enterprises. 1995. AppBuilder. http://www.appbuilder.com.

[46] John Maloney, Leo Burd, Yasmin B. Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. 2004. Scratch: A Sneak Preview. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing* (C^5). IEEE Computer Society, 104–109. https://doi.org/10.1109/C5.2004.1314376

[47] James Martin. 1981. *Applications Development Without Programmers.* Prentice-Hall.

[48] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT'14).* ACM, 103–104. https://doi.org/10.1145/2663171.2663188

[49] David May. 1983. OCCAM. *SIGPLAN Notices* 18, 4 (April 1983), 69–79. https://doi.org/10.1145/948176.948183

[50] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. 1962. *LISP 1.5 Programmer's Manual.* MIT Press.

[51] Charles H. Moore. 1974. FORTH: A New Way to Program a Minicomputer. *Astron. Astrophys. Suppl.* 15 (1974), 497–511.

[52] Johan Östlund and Tobias Wrigstad. 2010. Welterweight Java. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS) (LNCS, Vol. 6141)*, Jan Vitek (Ed.). Springer, 97–116. https://doi.org/10.1007/978-3-642-13953-6_6

[53] Rob Pike. 2012. Go at Google. In *Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, Gary T. Leavens (Ed.). ACM, 5–6. https://doi.org/10.1145/2384716.2384720

[54] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do — A Large-Scale Study of the Use of Eval in JavaScript Applications. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 6813)*, Mira Mezini (Ed.). Springer, 52–78. https://doi.org/10.1007/978-3-642-22655-7_4

[55] Eric Roberts. 2001. An overview of MiniJava. In *Proceedings of the 32rd SIGCSE Technical Symposium on Computer Science Education*, Henry MacKay Walker, Renée A. McCauley, Judith L. Gersting, and Ingrid Russell (Eds.). ACM, 1–5. https://doi.org/10.1145/364447.364525

[56] Frank Rubin. 1987. "Go To Considered Harmful" Considered Harmful. *Commun. ACM* 30, 3 (March 1987), 195–196. https://doi.org/10.1145/214748.315722

[57] SA22-7832-11. 2017. *z/Architecture Principles of Operation* (twelfth ed.). IBM.

[58] SA32-0978-00. 1988. *z/OS TSO/E CLISTs Version 2 Release 1.* IBM.

[59] SC09-2507-10. 2016. *Programming IBM Rational Development Studio for i. ILE RPG Programmer's Guide. Version 7.3.* IBM.

[60] SC14-7285-01. 2011. *Enterprise PL/I for z/OS Language Reference Version 4 Release 2* (second ed.). IBM.

[61] SC23-8528-01. 2009. *Enterprise COBOL for z/OS Language Reference. Version 4 Release 2* (second ed.). IBM.

[62] SC23-8529-01. 2009. *Enterprise COBOL for z/OS Programming Guide. Version 4 Release 2* (second ed.). IBM.

[63] Louis Schlueter. 1988. *User-Designed Computing: The Next Generation.* Lexington Books.

[64] Software AG. 2018. Facts about Natural for Mainframe. https://resources.softwareag.com/adabas-natural/2018-3-fs-natural-en-natural-mainframe-fact-sheet.

[65] Thomas Studer. 2001. Constructive Foundations for Featherweight Java. In *Proceedings of the International Seminar in Proof Theory in Computer Science, International Seminar (PTCS) (LNCS, Vol. 2183)*, Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk (Eds.). Springer, 202–238. https://doi.org/10.1007/3-540-45504-3_13

[66] Gerald Jay Sussman and Guy Lewis Steele Jr. 1975. *Scheme: An Interpreter for Extended Lambda Calculus.* Technical Report. MEMO 349, MIT.

[67] Andrey A. Terekhov and Chris Verhoef. 2000. The Realities of Language Conversions. *IEEE Software* 17, 6 (Nov./Dec. 2000), 111–124. https://doi.org/10.1109/52.895180

[68] TIOBE. 2019. The TIOBE Programming Community index. https://www.tiobe.com/tiobe-index/.

[69] Mads Torgersen. 2004. The Expression Problem Revisited. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 3086)*, Martin Odersky (Ed.). Springer, 123–143. https://doi.org/10.1007/978-3-540-24851-4_6

[70] Thi Mai Thuong Tran and Martin Steffen. 2010. Safe Commits for Transactional Featherweight Java. In *Proceedings of the Eighth International Conference on Integrated Formal Methods (IFM) (LNCS, Vol. 6396)*, Dominique Méry and Stephan Merz (Eds.). Springer, 290–304. https://doi.org/10.1007/978-3-642-16265-7_21

[71] Guido van Rossum. 1997. A Tour of the Python Language. In *Proceedings of the 23rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS).* IEEE Computer Society, 370. https://doi.org/10.1109/TOOLS.1997.10001

[72] Philip Wadler. 1998. The Expression Problem. Posted on the Java Genericity mailing list, http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[73] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. 1977. Prolog — The language and Its Implementation Compared with LISP. *SIGART Newsletter* 64 (1977), 109–115. https://doi.org/10.1145/872736.806939

[74] Wiki. 2013. Syntactically Significant Whitespace Considered Harmful. http://wiki.c2.com/?SyntacticallySignificantWhitespaceConsideredHarmful.

[75] Niklaus Wirth. 1971. The Design of a PASCAL Compiler. *Softw., Pract. Exper.* 1, 4 (1971), 309–333. https://doi.org/10.1002/spe.4380010403

[76] Donald R. Woods and James M. Lyon. 1973. *The INTERCAL Programming Language Reference Manual.* https://www.muppetlabs.com/~breadbox/intercal-man/.

[77] Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 212–218. https://doi.org/10.1145/3136040.3136058

[78] Vadim Zaytsev. 2020. Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler. *Proceedings of the 16th European Conference on Modelling Foundations and Applications in the Journal of Object Technology (ECMFA@JOT)* 19 (July 2020), 5:1–22. Issue 2. https://doi.org/10.5381/jot.2020.19.2.a5

[79] Tian Zhao, Jens Palsberg, and Jan Vitek. 2003. Lightweight confinement for featherweight Java. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages and Applications*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 135–148. https://doi.org/10.1145/949305.949318