Language Design with Intent

Vadim Zaytsev (http://grammarware.net), Raincode Labs, Brussels, Belgium

***Abstract*—Software languages have always been an essential component of model-driven engineering. Their importance and popularity has been on the rise thanks to language workbenches, language-oriented development and other methodologies that enable us to quickly and easily create new languages specific for each domain. Unfortunately, language design is largely a form of art and has resisted most attempts to turn it into a form of science or engineering. In this paper we borrow concepts, techniques and principles from the domain of persuasive technology, or wider yet, design with intent — which was developed as a way to influence users behaviour for social and environmental benefit. Similarly, we claim, software language designers can make conscious choices in order to influence the behaviour of language users. The paper describes a process of extracting design components from 24 books of eight categories (dragon books, parsing techniques, compiler construction, compiler design, language implementation, language documentation, programming languages, software languages), as well as from the original set of *Design with Intent* cards and papers on DSL design. The resulting language design card toolkit can be used by DSL designers to cover important design decisions and make them with more confidence.**

I. Motivation

First software languages were used in late 1940s[1] as an intermediate step in algorithm design. They allowed programmers of digital computers to bridge the gap between mathematical computations and machine codes. (The codes as such are much older, they were used on punched cards and rolls since 1725 in weaving looms[2] and at least since 1842 in pianolas[3].) A decade later[4] people started developing automated compilers, delegating the task of translating texts written in these languages, to the machine code, to system software components. Another decade passed, and new languages started being developed with specific design aims, targeting a particular problem domain[5] or a particular target audience[6]. By 1969 there were at least 120 widespread software languages [15], [35]. The next two or three decades, the language landscape was becoming more and more populated and — some claim — cluttered with numerous languages designed and implemented for all kinds of goals and purposes. Eventually we all have arrived at the point where creating a new language suitable for the problem at hand, ceased being challenging for engineers. Having, reusing or designing a DSL has been elevated to just a regular MDE problem solving recipe. Now we are focused on making software language creation methods reliable and repeatable [36].

[1]Since von Neumann and the Goldstines' *Flow Diagrams*.
[2]Since Basile Bouchon's silk centre in Lyon.
[3]Since Claude Félix Seytre's French patent no. 8691.
[4]Since Hopper's *MATH-MATIC*.
[5]Since Iverson's *APL*.
[6]Since Papert's *LOGO*, strengthened later by Perlman's *TORTIS*.

In this paper we assume the standpoint of *software language engineering* and, whenever possible, make no explicit distinction between modelling languages and programming ones, between domain-specific and general-purpose ones, among generations, paradigms, etc. Thus, whenever possible, we say "user" or "language user" instead of "programmer" or "modeller", and use other kinds of neutral terminology. We use the word "model" instead of "language instance" to mean a model, a program, a query, a stylesheet, a spreadsheet, etc. Other principles behind this project are explained in section II.

Languages are designed for following purposes, a.o.:

- to raise the abstraction level (almost universal);
- to improve user experience for languages with known problems but infeasible evolution (C++ for C, Dart for JavaScript, Go for C++, Swift for ObjectiveC, Scala for Java, Hack for PHP, .NET Core for .NET Framework);
- to give domain experts control over executable systems (the goal behind most domain-specific languages);
- to let non-coders structurally communicate with computers (emojis and smileys in most social networks, web forum markup like bbcode, wiki markup, etc);
- to open the usage of tools and services for third party usage (APIs);
- to abstract from irrelevant boilerplate (combinator libraries, languages with built-in constructs for concurrency, error handling, design patterns, etc);
- to explore different ways of human-computer interaction (numerous spreadsheet applications, most languages developed in workbenches like MPS or MetaEdit+);
- to make expressive and robust interchange and storage formats (even JSON and XML work with schemata);
- to build efficient tools by choosing suitable data structures (intermediate representations);
- to redesign legacy languages (VB.NET aligned with C#, XHTML as HTML in XML);
- to evolve existing languages into new versions (coevolution of Java and C# since the initial release of the latter);
- to create attractive language dialects (several industrially applicable extensions of originally educational Pascal, many vendor-specific COBOL compilers incompatible among themselves to prevent users from migrating);
- to experiment with new paradigms and get to know limits of their expressiveness (bidirectional transformation, reversible computation and others).

However, "language design is largely an art, not a science" [11, p.67]. There is no clear separation of where the language design starts and where it ends. In practice the work of a software language designer often gets mixed with the

work of a requirements engineer or a tool implementer. In theory, books with "compiler design" in the title (such as the new series by Springer) are filled with minute implementation details and contain almost no information on design decisions and problem/solution modelling. To clarify the issue, we turn to professional designers and their theories.

Interaction designers are people for whom "the purpose of the profession [is] to change the way people behave" [37]. Essentially, designers of software languages (languages for programming, modelling, specification, configuration, markup, etc) are the same: they create environments and ecosystems for their end users (modellers, programmers, etc) to express themselves in a particular way. One of the current trends there is to stimulate and enforce sustainable changes in people's behaviour [28]–[31]. We follow them by treating "sustainable" in a broad sense: we want to acknowledge that introducing languages in practice not only forces people into writing or drawing their models in a particular way, but also teaches them how to reach their goals better and gradually turn language usage principles into patterns of the trade. To do this, we particularly investigate the interaction design method called *Design with Intent* [25]–[27], along with its applications [28], [29] and reports on its development [30]–[34].

For some designers and sustainability researchers it is unclear where the border lies between systems that are designed to gently guide users' behaviour and automatic systems that bypass users' decision making altogether [25], [38]. Others introduce a spectrum for distribution of control that goes from Informing to Feedback to Enabling to Encouraging to Guiding to Seducing to Steering to Forcing to Automating [39] or provide two-dimensional models of influence with a separate axis for *force* (weak to strong) and for *salience* (hidden to apparent) [40]. The state of the art in language design is to stick to the extremes and either employ fully automated methods of grammar inference [41], grammar recovery [42], language identification [43], etc, or go for the "art, not science" mode. Interestingly, IDE development is one step ahead, and there are many plugins that support development in chosen languages by (using the terminology of [39]) providing feedback or enabling users to perform some actions automatically only if they explicitly decide so.

With respect to time pressure, industrial DSL designers (this paper's author included) are somewhere between interaction designers and chess players, which (together with nurses, firefighters, military personnel and nuclear plant operators) were observed to profit greatly from the *recognition-primed decision model* [44]. Stimulating it deliberately is known as *naturalistic decision making*. Drawing from the chess domain further, we have the term *gambits* to describe the repertoire of principles and mental models that experienced designers and architects bring with them to be quickly pattern matched to propose concrete solutions to the problems at hand [45], [46]. However, it has been known for a long time that one particular inventor or designer usually works within a trend and is naturally limited by his or her memory for inspiration [47]. Hence, we have an abundance of books, approaches and methods pushing toward *lateral thinking*, which are too numerous to cite. One exception is useful to note here: the power of *brainstorms* has been apparently largely exaggerated, since scientific attempts to investigate it have shown that group interactions and phenomena like social loafing (the more people, the less accountable each person feels) [48], evaluation apprehension (the more people, the greater the worry that ideas will be received poorly) [49] and production blocking (the more people, the lower the chances of each contributing well) [49] lead to brainstorming groups collectively producing fewer, less valuable and less diverse ideas than the individuals would have produced on their own [50]. In contrast, stimulating lateral thinking and *divergent production* by using design toolkits is a widely used and acknowledged method — Lockton [25] provides a great overview of them.

The **problem** we will be solving in this paper is *to provide software language engineers with a set of concrete actionable guidelines in the form of a design toolkit*. Each element of such a toolkit may or may not be relevant to a particular design endeavour, but if it is, it should provide a short description of the impact of the element on the future language users, as well as on the rest of the design of this language. The main motivation behind our solution is a psychological theory of intelligence as a set of teachable strategies [51]–[53].

Designers outside software make choices that are solidified in the resulting products in a way that makes it often obvious to trace back to them: if a button is big and red, it was deliberately made so prominent to emphasize how important and dangerous it is; if there are spikes on the place that otherwise looks suitable for sitting, they have been put there deliberately to avoid you sitting down; etc. In language design, these are two different worlds which are often disjoint: most programmers do not know why their languages have semicolon-terminated statements, they do not think about reasons that led to the appearance of a garbage collector in their language, and they do not have to know whether the compiler for their code has a top-down or a bottom-up parser inside it, or whether the language semantics is specified denotationally or operationally. This internal design also has some principles like separation of concerns or parsing techniques, which have entire books and thousands of articles behind them, but in this particular project we focus our efforts on those aspects of language design that impact the language users. Since we want to change or influence the behaviour of language users, we should a priori filter out aspects that are invisible to them.

This approach allows us not only to conceptually reuse results from sustainable behaviour research [28]–[31], [38], but also to formally separate language **design** from both language construction and compiler design, which have been mixed and intertwined in all previously existing literature. *We classify any aspect of a software language as a design aspect, if and only if the choices around it will directly affect the behaviour of the language users*, and classify it otherwise if they will not. In that sense, for instance, bootstrapping (implementing parts of the compiler of a language in that language) is irrelevant for design, even though it is an interesting construction problem.

II. Research Principles

The final result of this project—a software language design toolkit—is available at http://slebok.github.io/dyol and is highly recommended for closer inspection, since the paper space constraints do not allow to do justice to all the cards. The following three sections will describe the process of creating it. It will be constructed based on the following three principles:

No discrimination: we will cover sources about DSLs, GPLs, DSMLs, UML, API, and anything that looks like it could be related to computer languages and their design.

No prejudice: we will avoid any bias as far as possible, not lean towards any particular paradigm and not label any design choices as problematic or harmful, as often done in prior research.

No metadata: we will collect as many design elements as possible, from as many sources as remains feasible, but will not inherit any classification schemes (such as Lockton's lenses [25], [27]–[29], [34] or Wile's design concerns [54] or Mernik et al.'s DSL development phases [55]) from them.

III. Research Process

To form an initial language design deck, we have gone through all the 101 cards of *Design with Intent* [26] and tried to map each of them into the software linguistics technological space. This is a well-known modelling/borrowing technique that has become classic in some areas: for example, defining *smells* in a new domain is almost exclusively done by going through the program smells in the book that proposed them [56], filtering out irrelevant ones and adapting the rest to fit the domain. However, in our case we decided to mind the gap between the domains of interaction design and language design, and refine the result with the corpus of books from these eight categories (later their names are abbreviated):

- *Dragon Books* — the classic, traditional studybooks; while getting inevitably outdated, they are strongly associated with the domain of language processing.
- *Parsing Techniques* in a narrow, text-to-model sense. Unlike parsing in a broad sense [57] which is closer related to SLE and MDE, these books are expected to contribute little to the toolkit.
- *Compiler Construction* — mostly course books for students learning how to make language translators.
- *Compiler Design* — books claiming to explain external and internal design details around language translators.
- *Language Implementation* — similar to the previous two, but with more focus on practical aspects and implementation efficiency.
- *Language Documentation* — documents explaining one particular language to its potential users.
- *Programming Languages* — collections of concepts found in existing languages.
- *Software Languages* — books with general applicability, not limited to DSLs or programming.

Practically, the books (shown in Table I and summarised in section IV) were chosen from the private library of Raincode Labs, an industrial company specialised in compiler services; most of them are used more or less regularly in everyday compiler writing. Several extra books were purchased following advice by the colleagues of the paper author, all compiler experts. In the interest of time, since (re)reading one such book was taking up to a week, when faced with choices, we were preferring books within the same category that were versatile with respect to style or publication date.

Per book, the following steps were taken:

- Go through the book contents, try to identify already known cards as well as new ones, mark page numbers.
- Go through the marked elements (cards), identify duplicates among those that are too similar.
- Go through the elements remained unmarked in this book, check if they map to anything newly observed.

Every now and then we filter out design aspects that are too universal. For instance, version control systems are definitely important for the process of software engineering, but they are way beyond the control of a language designer. Certain aspects of their use can form a natural part of the language ecosystem — those are decomposed and included individually for related parts. For example, Go's standard format convention is a combination of using commit hooks (a candidate card but not a card in the final version) and Pretty-printing (a proper language design card; from now on they will be presented in this font and colour).

It should be noted here that the original idea was using the index of terms at the end of each book. It worked extremely poorly. These indices are of very varying quality and degree of granularity, the terminology is contextual and oscillates over the years. In the end all the markings were made manually, also to ensure fair treatment of synonyms and near-synonyms: e.g., "identifiers" in PT books were mostly related to Variables, "interpreting automata" are a way of looking at Runtime, by "substitution" some books mean substitution of arguments into procedure parameters (i.e., Parameter Passing) and use "coersion" for Substitution (of entities of one type by entities of another type), parsing concurrently does not mean coverage of Concurrency within the language, and the definition of a formal grammar of string languages must be marked as Concrete Syntax even if the words "concrete syntax" do not appear anywhere in the book.

After the cycle was completed for all books, we had to go through it once again in a quicker mode to mark cards that were identified later to books that were processed before that.

Finally, we processed three sources that we consider conceptual ancestors of this project. Wile published a paper on the DSL spectrum [54], which contains a lot of motivation, but also (in section 4) several flexibly organised language design concerns, issues and problems. For each of them we considered whether they have any impact on language users — in which case the name of the issue was noted as well. Spinellis made an attempt to start collecting design patterns for DSLs [58], which succeeded in staying relatively technology-agnostic, unlike later publications on language implementation patterns [59]. Four of them were added to our toolkit's

Dragon Books	DB-GD	*Principles of Compiler Design*	1977	[1]
	DB-RD	*Compilers: Principles, Techniques, and Tools*	1986 (1 ed)	[2]
	DB-PD	*Compilers: Principles, Techniques, & Tools**	2006 (2 ed)	[3]
Parsing Techniques	PT-AO	*Definition of Programming Languages by Interpreting Automata**	1974	[4]
	PT-HU	*Introduction to Automata Theory, Languages, and Computation*	1979	[5]
	PT-GJ	*Parsing Techniques: A Practical Guide*	2008 (2 ed)	[6]
Compiler Construction	CC-DG	*Compiler Construction for Digital Computers*	1971	[7]
	CC-WG	*Compiler Construction*	1984	[8]
	CC-NW	*Compiler Construction*	2005	[9]
Compiler Design	CD-AH	*Compiler Design in C*	1990	[10]
	CD-SM	*Advanced Compiler Design and Implementation*	1997	[11]
	CD-GR	*Modern Compiler Design*	2012 (2 ed)	[12]
Language Implementation	LI-BH	*Brinch Hansen on Pascal Compilers*	1985	[13]
	LI-RM	*Writing Compilers and Interpreters: An Applied Approach*	1996	[14]
	LI-PZ	*Programming Languages: Design and Implementation**	2001 (4 ed)	[15]
Language Documentation	LD-ED	*A Primer of ALGOL 60 Programming*	1962	[16]
	LD-JW	*Pascal User Manual and Report*	1985 (3 ed)	[17]
	LD-WH	*Programming in the .NET Environment*	2003	[18]
Programming Languages	PL-BM	*Principles of Programming Languages: Design, Evaluation and Implementation*	1983	[19]
	PL-WC	*Comparative Programming Languages*	1993 (3 ed)	[20]
	PL-RS	*Concepts of Programming Languages**	2001 (5 ed)	[21]
Software Languages	SL-AS	*Structure and Interpretation of Computer Programs*	1996 (2 ed)	[22]
	SL-CF	*Engineering Modeling Languages: Turning Domain Knowledge into Tools*	2017	[23]
	SL-RL	*Software Languages: Syntax, Semantics, and Metaprogramming*	2017	[24]

TABLE I

SOURCES: THE 24 BOOKS CLASSIFIED IN 8 CATEGORIES. THE THREE BOOKS WHOSE TITLES ARE MARKED WITH AN ASTERISK, ARE CODED BY THEIR CORRESPONDING RUSSIAN TRANSLATIONS (PAGE NUMBERS IN THE ORIGINALS MIGHT DIFFER).

links, the other four were assessed to belong to language construction. Finally, there is the well-cited[7] *When and How to Design DSLs* [55], which introduces five development phases of a language: decision, analysis, design, implementation and deployment, lists several patterns for each of the first four phases, and gives examples of those patterns used in real languages. These patterns are of varying usefulness for us: the decision ones are definitely what we classify as language design; the analysis patterns are way too abstract to be of any use, half of the design patterns are the same and the rest are mapped to one Sublanguage card, some implementation patterns are marked, and Deployment was marked as the entire phase since the paper fails to elaborate on it.

The resulting language design card toolkit called **DYOL** (pronounced as "duel"), for **D**esign **Y**our **O**wn **L**anguage, can be found summarised on Table II and in complete form with 96 cards at http://slebok.github.io/dyol.

IV. THE CORPUS

In this section we introduce the corpus shown on Table I by dedicating one paragraph to each of the categories to summarise the books.

There are three Dragon Books: the Green, DB-GD [1], the Red, DB-RD [2] and the Purple, DB-PD [3], each next one a direct extension of its predecessor. Their nicknames come from the dragons depicted on covers of most editions, with "complexity of compiler design" written on the beast being attacked by a knight with names of various compiler engineering techniques inscribed on its armour and weapons. Dragon Books are equally popular as self-study material, as university course foundations and as cookbooks among industrial developers. The most popular one is DB-RD, since DB-PD has sacrificed many detailed discussions on topics of general importance and, instead of adding object orientation, added several hundred pages on code optimisation, in particular related to parallel computation. There is a big audience for those, but they are not every compiler engineer's concern.

PT-AO [4] is an ode to the Vienna development method, and is a side product of a project of modelling the semantics of PL/I in a formal way. It does not contain a lot of useful information on design, but the entire category of parsing books was included in our corpus to confirm that parsing in the narrow sense and language design are unrelated. This particular book is pretty realistic and mixes formal definitions with a selection of language constructs which a real world language designer may decide to use. It also covers Concurrency and Synchronisation, but surprisingly avoids the complexity of Parameter Passing discussion when formalising Subprograms. PT-HU [5] is better known as the Cinderella Book for its artistic cover, and was the rewrite of the original *Formal Languages and Their Relation to Automata* [60]. It restructured the material while also made it more mundane by cutting away treatment of deeper issues. Despite the critique, even after all these years it is a great book to learn formal language theory and the basics of the theory of computation, but it has precious little to do with language design. Neither PT-AO nor PT-HU mention Operator Precedence when talking about ambiguous grammars. PT-GJ [6] is the bible of parsing, a very dense description of most, if not all, parsing algorithms known to mankind up till 2008. As if that were not enough, the bibliography of the book is annotated, takes almost a hundred pages and turns several decades of parsing techniques research into several well-organised stories. There are only two downsides of PT-GJ: the existence of several significant advances made in the decade since it was last revised (GLL [61] and ALL(*) [62]), and the weak connection to language design.

[7] 1613 citations as of 19 April 2017, according to Google Scholar.

Access Modifier	Compilation Error	Esotericism	Keyword	Parametrised Type	Smell
Alphabet	Compilation Warning	Event	Labelling	Performance	Standard Library
Assignment	Comprehension	Exception Handling	Lazy Evaluation	Phased Process	Static Analysis
Backtracking	Concrete Syntax	Execution Error	Live Feedback	Picture Clause	Sublanguage
Backward Compatibility	Concurrency	Expressivity	Lock-out/Opt-in	Platform Lock-in/out	Subprogram
Block	Constraint	First Class Citizen	Macro	Pointer	Substitution
Branching	Cross-compilation	Garbage Collection	Metaphor	Pretty-printing	Synchronisation
Character Type	Debugging	Generation	Module	Preview	Syntactic Sugar
Class	Default	Heterogeneous Data	Natural Pattern	Product Line	Syntax Highlighting
ClientServer	Deployment	IDE	Numeric Data Type	Readability	Trade Off
Code Completion	Deprecated Construct	IDE GUI	Operator Overloading	Record	Type Analysis
Code Generation	Design ChartDiagram	Indentation & Whitespace	Operator Precedence	Redefine	Type Definition
Code Mining	Documentation	Inheritance	Optimisation	Refactoring	Undefined Behaviour
Code Ownership	Encapsulation	InputOutput	Order	Runtime	Unification
Collection	Energy Saving	Instruction Set	Orthogonal Design	Scope & Binding	Variable
Comment	Enumeration Type	Iteration	Parameter Passing	Security	Virtual Machine

TABLE II
THE NAMES OF THE 96 CARDS OF DYOL V0.2, IN LEXICOGRAPHICAL ORDER.

CC-DG [7] had the same domain-defining status before the first Dragon Book came out. It is still useful now (46 years later!) but some terminology has noticeably changed in the meantime. CC-WG [8] is true to its name: most elements of a language related to *design* or any part visible to the user, are enumerated quickly in the introduction, and the main focus of the rest of the book is on the implementation details and in how to construct a compiler in the best possible way. One of the interesting oddities is that this book never talks about Comments, even in the lexical analysis chapter. CC-NW [9] is rather minimalistic and refuses to acknowledge modern language concepts such as Class even though the version we processed was updated by the author in 2005.

CD-AH [10] is heavily implementation-driven, and shows everything from the point of view of the compiler writer. CD-SM [11] has "advanced" in the title and indeed starts off on the advanced level (e.g., Type Definition comes before Variable declaration), Subprogram calls are considered in much more detail beyond Parameter Passing (prologues, epilogues, returns, etc); several chapters dedicated to optimisations, like in DB-PD. CD-GR [12] is currently (as of 2017) one of the most comprehensive guides to the compiler world, recently updated to include new research achievements. It is linked from 53 DYOL cards, this exceeds the average number twofold.

LI-BH [13] is a book about compilers for beginners. It takes the readers by the hand and walks them very gently through all phases of designing and developing a compiler of a subset of Pascal written in a superset of Pascal. Some terminology has become outdated: e.g., Keywords are "word symbols" and Virtual Machines are "language-based computers". The author also heavily optimises every step which has a fair chance of teaching people premature optimisation, and makes some sections unnecessarily outdated for dismissing certain design options for the compiler because they would require 82kb more memory than the personal computer could have. Besides those points, it is a great introductory resource for writing language translators. LI-RM [14] contains a story of developing a manually written compiler in C for a fairly advanced Pascal-like language. It starts off very basic and gradually acquires more realistic and useful details. LI-PZ [15] is structured top-down from the programming language concepts towards reviews of existing languages.

LD-ED [16] was an interesting challenge since its style is different from both compiler books and from modern language documents. Sections have unhelpful Dijkstraesque names: "local quantities" mean the Order of declarations, "formal parameters" is about Scope & Binding, etc. Interestingly, Character Type (strings) are neither mentioned nor used in this book, except in the appendix which contains Backus et al.'s *Revised Report*. LD-JW [17] explains Pascal the way its inventor thought perfect: by showering the reader with syntax diagrams. LD-WH [18] is one of the many modern language manuals we have on our desks, and was chosen because it presented not just one language out of its context, but an entire platform. The second half of the book is dedicated to specific languages that run on .NET (Visual Basic, C#, Python, Perl, Active Oberon, Component Pascal, HotDog and Mondrian). Because of this platform focus, the book covered many language design aspects that were otherwise easily overlooked (Runtime, Standard Libraries, Deployment, etc).

PL-BM [19] has the same name (*Principles of Programming Languages*) as the flagship conference in the domain of programming languages, but is not as overly formal as their proceedings. It gives a fairly comprehensive overview of the domain for early 1980s and includes concepts such as OOP and Garbage Collection. PL-WC [20] covers more concepts and more languages and profits fully from the decade that separates it from PL-BM. PL-RS [21] is fairly similar, but unfortunately not many breakthroughs were made in the meantime, so in general its added value for those familiar with PL-BM and PL-WC is minimal.

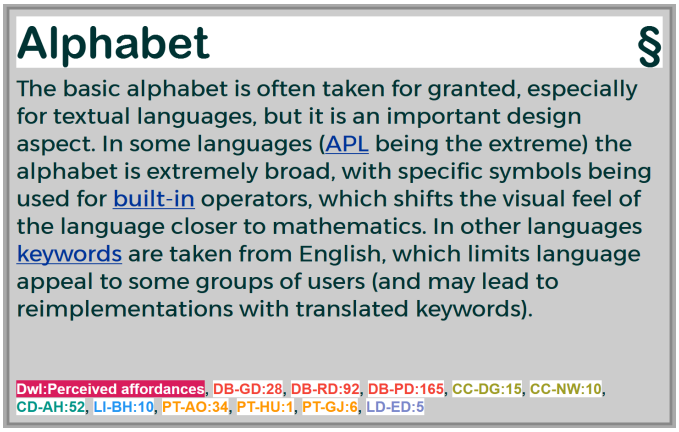
SL-AS [22] is a very famous book that is not directly associated with software language engineering, but the principles explained there (like modelling with data), are universally applicable. The book covers a wide range of topics, is written in a reader-friendly style and is generously sprinkled with realistic examples — this combination brought it its popularity. SL-CF [23] is perhaps the most significant book of this year in MDE and covers many concepts and technologies of creating domain-specific modelling languages. SL-RL [24] has not

been published yet, but a final draft was requested from the author and granted specially for this project. When published, this book will fill the niche of a standard studybook for SLE courses [63]. At this point, it was particularly hard to map to our cards because of its ambitious nature: many notions like Variables, Classes or Inheritance are used without ever explaining or introducing them, and many others (much more than were put on cards as of yet) are mentioned in passing. In the end we confirmed marking only those concepts that either took considerable space or were explained as being significant.

V. DYOL: The Language Design Toolkit

The resulting **DYOL** (**D**esign **Y**our **O**wn **L**anguage) deck can be found at http://slebok.github.io/dyol. Its snapshot from the time of paper's submission is archived with doi:10.6084/m9.figshare.5234581.v1.

As an example, consider a card called Alphabet:



The textual description quickly introduces the design aspect. The text refers to external resources (http://tryapl.org) and to two other cards (Standard Library and Keyword). On the bottom, we have a list of markings for this card. The only Design with Intent relation is "perceived affordances" from the perception lens (indicated by the pink background colour) which describes how the form of some elements of the system being designed can suggest or constrain particular actions of the end users. Just like reshaping the waste recycling bins was shown to increase recycling levels, choosing the alphabet to support or prefer bare numbers, card holes, Unicode, ASCII, EBCDIC, emoji, etc, suggests what one can do with the programs in such a language, on which platform that should happen and who is the target audience (e.g., mathematical notation would suggest scientists and colourful cartoon icons suggest children). The other markings show that the notion of an alphabet is always introduced up front in parsing books and is mentioned eventually in compiler implementation-biased books. The § symbol in the top right corner of the card refers to an individual card page at http://slebok.github.io/dyol/cards/Alphabet.html which contains concrete examples of punch cards, character tables, fan-made APL keyboards and translated Concrete Syntaxes. The individual page also goes slightly deeper into the subject by linking the notion of an alphabet to the notion of a *dictionary* (the typical way of higher level language designers to call their alphabets to stress the fact that atomic language elements are words and not just letters of symbols).

DYOL v0.2 contains 96 such cards (recall Table II). They were challenging to make, but they are also obviously challenging to consume in one sitting, so some grouping and classification of them is needed. Let us try an automated approach first. By having or not having a mark that a particular card is handled by a particular book, we get a matrix of 96 by 24 of Boolean values. Compressing it by book category, we get 96 vectors with 8 values each (each value is between 0 and 3, inclusive). Running k-means clustering analysis on this data produced by the prevalence of marks per category of books, yielded one big cluster of 40 cards that were mostly never marked, as well as the following four (cluster names added manually):

- **Programming Core**: Assignment, Branching, Collection, Iteration, Pointer, Record, Scope & Binding, Subprogram, Type Definition, Variable.
- **DSL Core**: Comment, Constraint, Encapsulation, Event, Heterogeneous Data, IDE, Indentation & Whitespace, Input / Output, Instruction Set, Labelling, Lazy Evaluation, Numeric Data Type, Pretty-printing, Substitution, Virtual Machine.
- **Compiler Practice**: Alphabet, Block, Code Generation, Compilation Error, Compilation Warning, Concrete Syntax, Design Chart/Diagram, Keyword, Operator Precedence, Optimisation, Phased Process, Runtime, Static Analysis, Type Analysis.
- **Language Processing**: Backtracking, Character Type, Class, Concurrency, Debugging, Enumeration Type, Exception Handling, First Class Citizen, Garbage Collection, Generation, Inheritance, Macro, Module, Parameter Passing, Redefine, Synchronisation, Unification.

The **Programming Core** cluster contained cards marked in all categories besides PT (which was low on marks anyway) and SL (which, we presume, intentionally glanced over topics so prominent elsewhere). The **DSL Core** cluster was formed around cards marked mostly in SL with a strong presence in some combination of LI, LD and PL. The **Compiler Practice** cards were strong in DB, CC, CD and LI. The **Language Processing** cards were strongly present in PL, almost absent in PT and CC but never totally out of the other categories.

This is a very crude way of classifying the cards, but it forms a plausible starting point to introduce a classification like Lockton's *lenses* [27], [28], [34], Wile's *concerns* [54] or Mernik et al.'s *phases* [55], but grounded solidly on the marks made in existing information sources. The result needs either some manual tweaking (such as making sure that all the data types end up in the same group) or more base data.

Alternatively, we could contemplate explicit modelling of the users, as was also done for Design with Intent [29], to map behaviour patterns of language designers to pre-cooked subcollections of cards. This could be validated by looking at designs of existing languages and decomposing

them into cards, effectively reverse engineering the designer's decisions. This future direction, if pursued, is expected to yield several recognisable categories like "concrete", "abstract", "statics", "dynamics", "integration", etc. It is also quite likely to face challenges of explaining past design's imperfections. For example, in Java, Access Modifier and Defaults form an antipattern, since having no modifiers results in a state unreachable by any other modifiers (subclasses of the same package will have access to the field but subclasses from other packages will not).

At the present moment, each card has a very short explanation, possibly linking to other cards or external resources and providing hover help on acronyms. Prior to submission of this paper, the toolkit was seen only by a selected few people (not exclusively compiler engineers) who helped to refine the definitions. Ideally, each card should get its detailed individual page with an extended description akin to how design patterns are described, with consequences spelt out and with links to related papers that treat each topic in depth. The Alphabet example shown above is one of the few reasonably complete individual pages, but even that one can be improved.

For example, Van Tassel conducted a thorough historical investigation on the use of Comments from the beginning of programming to the present day [64], which helps put them into perspective and allows designers to understand why most software languages maintain the strange tradition of mixing documentation with implementation, and why computing pioneers like Knuth had such strong opinions on the matter [65]. Documentation itself was at some point meticulously metamodelled by Zaytsev and Lämmel [66]. Similarly, Martini investigated the history of types and explained in detail how implementation concerns such as "mode of calculation" gradually were recognised as design concerns and yielded an entire field of research around Type Analysis [67]. Van Roy maintains a diagram of the principal programming paradigms [68] which explains how logic programming is made from functional programming by adding Unification, and in turn becomes something else after adding Constraint solvers, Concurrency and finally Synchronisation, and one of our future projects involves aligning his programming landscape megamodel with DYOL. Together with collaborators, Van Roy showed how this Labelling is of great use in teaching programming [69], which was also the experience of the author [74]. Augmenting the DYOL elements with links to such detailed investigations will add depth and credibility to the toolkit as a whole, as a means to establish importance of each element.

VI. Conclusion and Future Work

By using the available material on development of the *Design with Intent* method and its results [25]–[34], and by marking (*coding*, in social science's terminology) books that are seen to be related to language design [1]–[24], we have formed a grounded theory of software language design. Following interaction design theory and persuasive technology research, we define *design* as something that changes behaviour of the users. With this definition, we identified 96 cards with 811 marks total, and presented them as a software language design toolkit that can be used by API developers, DSL makers, format designers, etc (http://slebok.github.io/dyol). The larger scope of this project is SLEBoK as an attempt to move towards a SWEBOK-like [73] book of knowledge for software language engineering, which is one of the current open problems of SLE [36]. Examples of other subprojects of SLEBoK are BibSLEIGH (http://bibtex.github.io) which collects thousands of research papers related to SLE and analyses them [75], or GraSs (http://slebok.github.io/grass) which provides a classification/taxonomy of smells found in grammars, metamodels and other language definitions [70].

The obvious tactic of polishing the DYOL toolkit is to cover even more books to eliminate selection bias and other threats to validity. However, a more conceptually challenging step would be to cover more *categories* of them instead. Kinds of books not covered by this project in its current form but considered for the future, among others, are: multi-language and multi-paradigm books (between LD and PL categories) like *Seven Languages in Seven Weeks* or *Elements of Programming Style*; comparative language documentation (C# books for Java programmers, Swift ones aimed for ObjectiveC professionals, etc); coding advice books such as *Structured Programming*, *Code Complete* and *Clean Code*; PhD theses; blog posts. There is a tentative list of books to cover first, at http://slebok.github.io/dyol/books/wip.html. Some support from the community at some point would also be nice.

The cards used in software design, like CRC cards [71], are considerably more structured than the cards used in most physical design processes [25], so in the future we can possibly improve them significantly. The processes themselves are also much better structured as well, with contemporary design techniques like story-driven modelling [72] which would be great to learn to apply specifically to software language design. In any case the next step is to continue to follow in the footsteps of *Design with Intent* and to organise language design workshops where people use the DYOL toolkit to design languages and measure the degree of success. This plan is quite realistic and will allow to build up on previous similar activities of the author with teaching language design to product designers [76] and running compiler coding dojos at academic conferences [77].

Acknowledgements

The author would like to take a moment to show respect and appreciation to Raincode colleagues, especially Jean-Eric Waroquier who advised to have a look at LI-BH [13] and explained the role and importance of CC-DG [7]. Also, credit goes to Ralf Lämmel for providing a draft of SL-RL [24]. With the exception of his book, 10 books that ended up in the corpus came from the private library of the author, 6 from the library of Raincode Labs and 7 were purchased. Raincode Labs paid for these new books, as well as for at least 14 more, filtered out the final corpus for various reasons. The author also thanks Sabine Janssens for lending an artistic eye in proofreading and helping to polish the text on the cards.

References

[1] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, 1977.
[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
[3] A. V. Aho, M. S.-L. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Addison-Wesley, 2006.
[4] A. Ollongren, *Definition of Programming Languages by Interpreting Automata*. Academic Press, 1974.
[5] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
[6] D. Grune and C. J. H. Jacobs, *Parsing Techniques — A Practical Guide*, 2nd ed. Addison-Wesley, 2008.
[7] D. Gries, *Compiler Construction for Digital Computers*. JW&S, 1971.
[8] W. M. Waite and G. Goos, *Compiler Construction*. Springer, 1984.
[9] N. Wirth, *Compiler Construction*. Addison-Wesley, 2005.
[10] A. I. Holub, *Compiler Design in C*. Prentice-Hall, 1990.
[11] S. Muchnick, *Advanced Compiler Design and Implementation*, 1997.
[12] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen, *Modern Compiler Design*, 2nd ed. Addison-Wesley, 2012.
[13] P. B. Hansen, *Brinch Hansen on Pascal Compilers*. Prentice-Hall, 1985.
[14] R. Mak, *Writing Compilers & Interpreters: An Applied Approach*, 1996.
[15] T. W. Pratt and M. V. Zelkowitz, *Programming Languages: Design and Implementation*. Prentice-Hall, 2001.
[16] E. W. Dijkstra, *A Primer of ALGOL 60 Programming*, 1962, vol. 2.
[17] K. Jensen and N. Wirth, *Pascal User Manual and Report*, 1985.
[18] D. Watkins, M. Hammond, and B. Abrams, *Programming in the .NET Environment*. Addison-Wesley, 2003.
[19] B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation and Implementation*. Holt-Saunders, 1983.
[20] L. B. Wilson and R. G. Clark, *Comparative Programming Languages*, 2nd ed. Addison-Wesley, 1993.
[21] R. W. Sebesta, *Concepts of Programming Languages*. Pearson, 2001.
[22] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, 1996.
[23] B. Combemale, R. W. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek, *Engineering Modeling Languages*. CRC Press, 2017.
[24] R. Lämmel, *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2017, in print. Marked version of 27 March 2017.
[25] D. Lockton, "Design with Intent: A Design Pattern Toolkit for Environmental & Social Behaviour Change," Ph.D. dissertation, Brunel, 2013. [Online]. Available: http://architectures.danlockton.co.uk/phd/
[26] D. Lockton, D. J. Harrison, and N. A. Stanton, *Design with Intent: 101 Patterns for Influencing Behaviour Through Design v.1.0*. Windsor: Equifine, 2010. [Online]. Available: http://www.designwithintent.co.uk
[27] ——, "The Design with Intent Method: A Design Tool for Influencing User Behaviour," *Applied Ergonomics*, vol. 41, no. 3, pp. 382–392, 2010.
[28] ——, "Exploring Design Patterns for Sustainable Behaviour," *The Design Journal*, vol. 16, no. 4, pp. 431–459, 2013.
[29] ——, "Models of the User: Designers' Perspectives on Influencing Sustainable Behaviour," *Journal of Design Research*, vol. 10, no. 1/2, pp. 7–27, 2012.
[30] ——, "Design for Sustainable Behaviour: Investigating Design Methods for Influencing User Behaviour," in *Proceedings of the 14th Conference on Sustainable Innovation*. The Centre for Sustainable Design, 2009.
[31] ——, "Making the User More Efficient: Design for Sustainable Behaviour," *Sustainable Engineering*, vol. 1, no. 1, pp. 3–8, 2008.
[32] ——, *Design with Intent: Persuasive Technology in a Wider Context*. Springer, 2008.
[33] ——, "Choice Architecture and Design with Intent," in *Proceedings of the Ninth Bi-annual International Conference on Naturalistic Decision Making*. British Computer Society, 2009.
[34] D. Lockton, D. J. Harrison, T. Holley, and N. A. Stanton, "Influencing Interaction: Development of the Design with Intent Method," in *Fourth Conference on Persuasive Technology*. ACM, 2009, pp. 5:1–5:8.
[35] J. Sammet, *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.
[36] A. H. Bagge and V. Zaytsev, "Open and Original Problems in Software Language Engineering 2015 Workshop Report," *SENotes*, vol. 40, 2015.
[37] J. Kolko, *Thoughts on Interaction Design*. Brown Bear, 2007.
[38] D. Lilley, V. Lofthouse, and T. Bhamra, "Towards Instinctive Sustainable Product Use," in *On Sustainability*, 2005.
[39] J. Zachrisson, G. Storrø, and C. Boks, "Using a Guide to Select Design Strategies for Behaviour Change," in *Proceedings of EcoDesign*, 2011.
[40] N. Tromp, P. Hekkert, and P.-P. Verbeek, "Design for Socially Responsible Behavior: A Classification of Influence Based on Intended User Experience," *Design Issues*, vol. 27, no. 3, pp. 3–19, 2011.
[41] A. Stevenson and J. R. Cordy, "A Survey of Grammatical Inference in Software Engineering," *SCP*, vol. 96, pp. 444–459, 2014.
[42] V. Zaytsev, "Notation-Parametric Grammar Recovery," in *LDTA*, 2012.
[43] J. Kennedy van Dam and V. Zaytsev, "Software Language Identification with Natural Language Classifiers," in *SANER'16*. IEEE, 2016.
[44] G. Klein, *Sources of Power: How People Make Decisions*. MIT, 1999.
[45] D. Perkins, *Knowledge as Design*. Lawrence Erlbaum Associates, 1986.
[46] B. Lawson, "Schemata, Gambits and Precedent: Some Factors in Design Expertise," *Design Studies*, vol. 25, pp. 443–457, 2004.
[47] T. W. Barber, *The Engineer's Sketch-Book of Mechanical Movements, Devices, Appliances, Contrivances and Details*. Spon, 1890.
[48] T. L. Robbins, "Social Loafing on Cognitive Tasks: An Examination of the 'Sucker Effect'," *Business & Psychology 9(3)*, pp. 337–342, 1995.
[49] M. Diehl and W. Stroebe, "Productivity Loss in Brainstorming Groups: Toward the Solution of a Riddle," *PSP*, vol. 53(3), pp. 497–509, 1987.
[50] A. Furnham, "The Brainstorming Myth," *Business Strategy Review*, vol. 11, no. 4, pp. 21–28, 2000.
[51] J. H. Flawell, "Developmental Studies of Mediated Memory," *Advances in Child Development and Behavior*, vol. 5, pp. 181–211, 1970.
[52] A. L. Brown, "The Role of Strategic Behavior in Retardate Memory," *Int. Review of Research in Mental Retardation*, vol. 7, pp. 55–111, 1974.
[53] J. Baron, "Intelligence and General Strategies," *Strategies in Information Processing*, pp. 403–450, 1978.
[54] D. S. Wile, "Supporting the DSL Spectrum," *Journal of Computing and Information Technology*, vol. 9, no. 4, pp. 263–287, 2001.
[55] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM CSUR*, vol. 37, no. 4, 2005.
[56] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design or Existing Code*. Addison-Wesley, 1999.
[57] V. Zaytsev and A. H. Bagge, "Parsing in a Broad Sense," in *MoDELS'14*, ser. LNCS, vol. 8767. Springer, 2014, pp. 50–67.
[58] D. Spinellis, "Notable Design Patterns for Domain-specific Languages," *The Journal of Systems and Software*, vol. 56, no. 1, pp. 91–99, 2001.
[59] T. Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic, 2010.
[60] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1968.
[61] E. Scott and A. Johnstone, "GLL Parsing," in *LDTA'09*, ser. ENTCS, vol. 253, no. 7, 2010, pp. 177–189.
[62] T. Parr, S. Harwell, and K. Fisher, "Adaptive LL(*) Parsing: The Power of Dynamic Analysis," in *OOPSLA'14*. ACM, 2014, pp. 579–598.
[63] A. H. Bagge, R. Lämmel, and V. Zaytsev, "Reflections on Courses for Software Language Engineering," in *EduSymp'14, CEUR 1346*, 2015.
[64] D. Van Tassel, "Comments," http://www.gavilan.edu/csis/languages/comments.html, 2004.
[65] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.
[66] V. Zaytsev and R. Lämmel, "A Unified Format for Language Documents," in *SLE'10*, ser. LNCS, vol. 6563. Springer, 2011, pp. 206–225.
[67] S. Martini, "Several Types of Types in Programming Languages," in *History and Philosophy of Computing*, vol. 487, 2015, pp. 216–227.
[68] P. Van Roy, "The Principal Programming Paradigms, v1.08," https://www.info.ucl.ac.be/~pvr/paradigms.html, 2008.
[69] P. Van Roy, J. Armstrong, M. Flatt, and B. Magnusson, "The Role of Language Paradigms in Teaching Programming," in *SIGCSE*, 2003.
[70] M. Stijlaart and V. Zaytsev, "Towards a Taxonomy of Grammar Smells," 2017, under review. [Online]. Available: http://slebok.github.io/grass
[71] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," in *OOPSLA*. ACM, 1989, pp. 1–6.
[72] U. Norbisrath, A. Zündorf, and R. Jubeh, *Story Driven Modeling*. CreateSpace, 2013.
[73] P. Bourque and R. Fairley, "Guide to the Software Engineering Body of Knowledge, Version 3.0," 2014, http://www.swebok.org.
[74] V. Zaytsev, "Flipped Top-Down is Systematic Bottom-Up," in *MoDELS EduSymp'15*, ser. CEUR, vol. 1555, 2015, pp. 17–28.
[75] ——, "BibSLEIGH: Bibliography of Software (Language) Engineering in Generated Hypertext," in *SATToSE'15, CEUR 1820*, 2017, pp. 54–64.
[76] ——, "The DSGA Model of DSL Design: Domain, Schema, Grammar, Actions," 2016, presented at DSLDI'16 at SPLASH.
[77] ——, "CoCoDo: Raincode Labs Compiler Coding Dojo," http://cocodo.github.io, 2017.