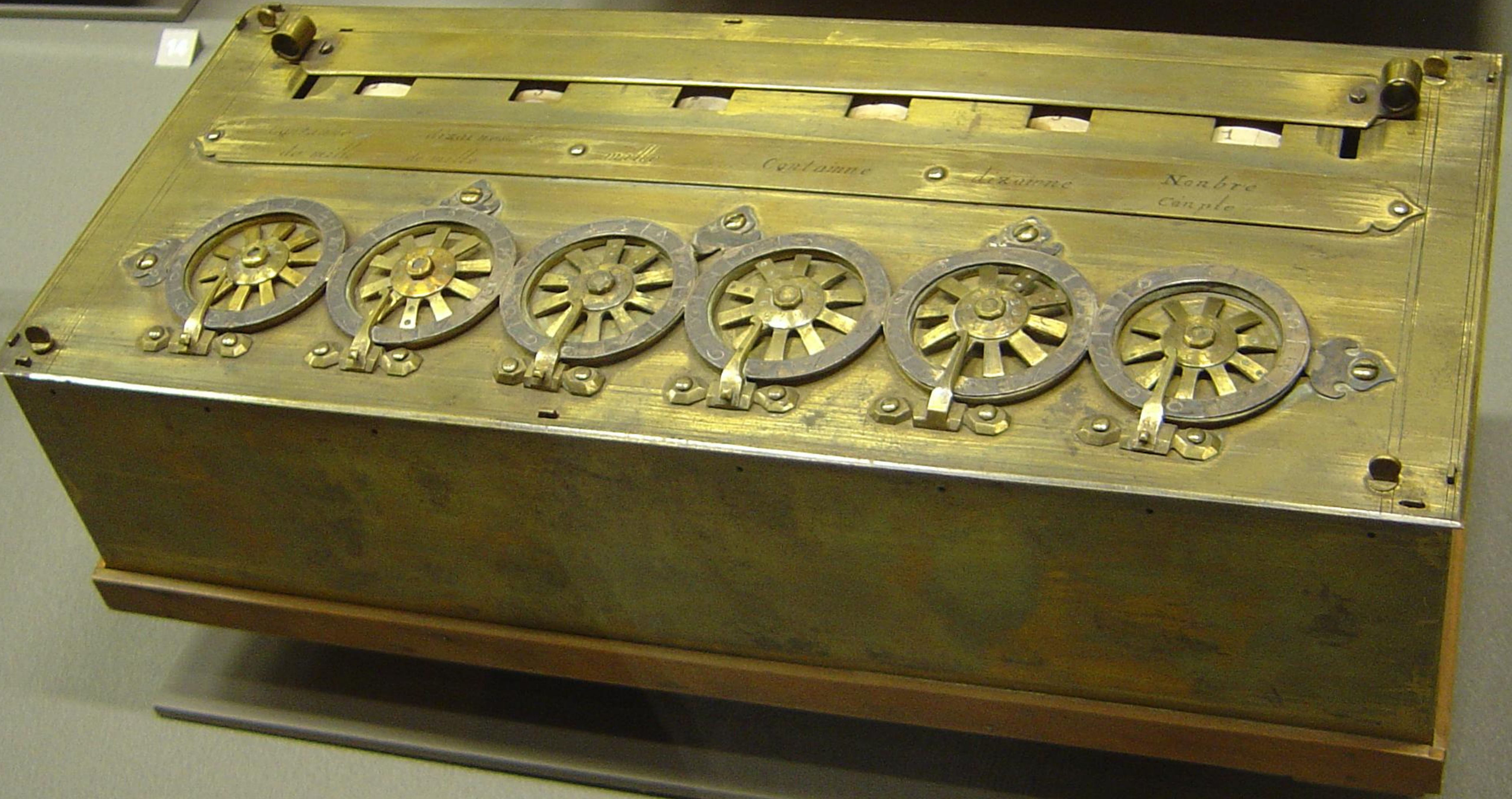
MODELS 2017

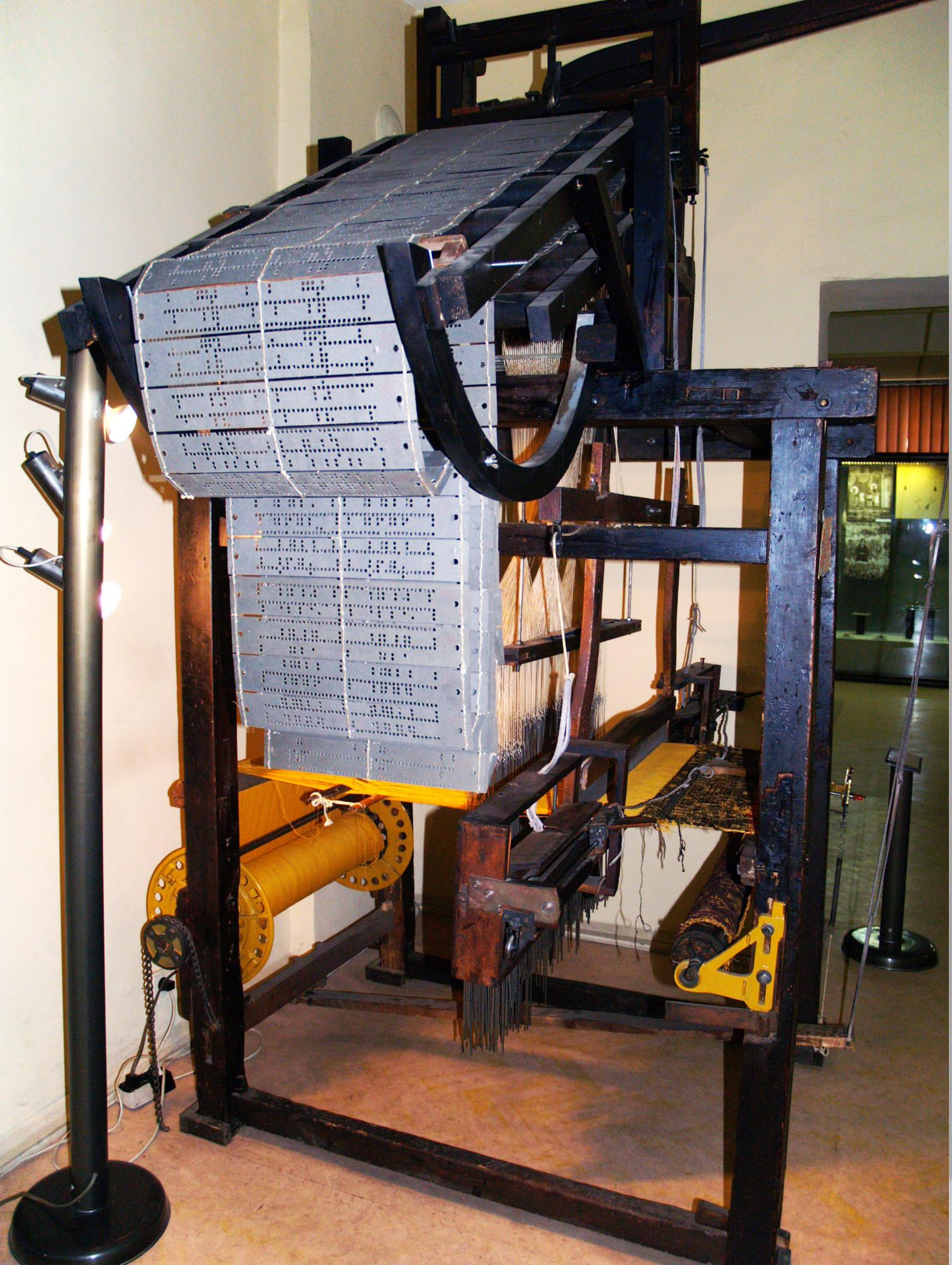
austin, tx

Software Language Design with Intent

or, How I Read 24 Books and Why

Dr. Vadim Zaytsev
CSO
raincode LABS
compiler experts


de mille
de mille
mille
Centainne
dezainne
Nonbre
Cenple
1



STECK
STECK
PIANOLA PIANO

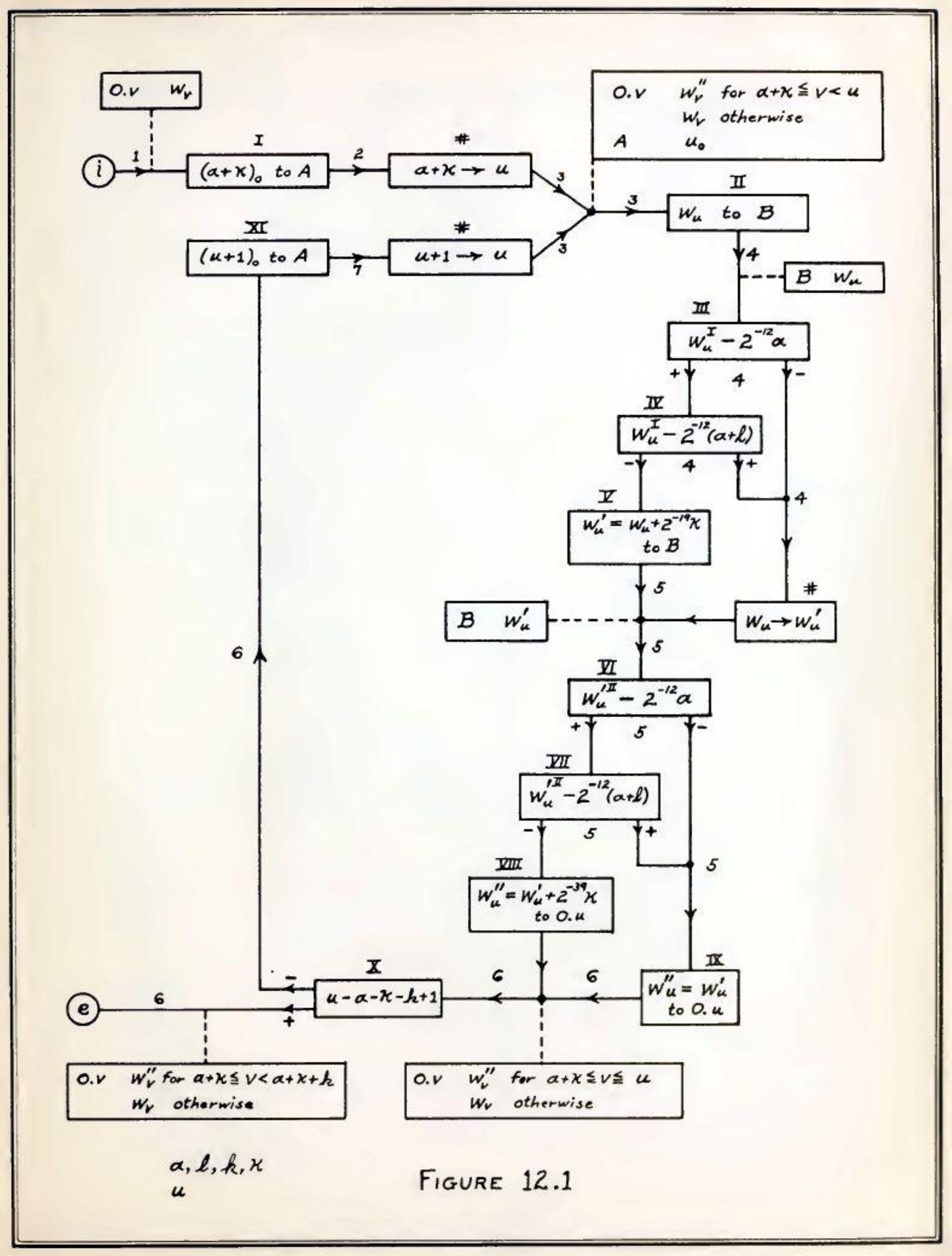
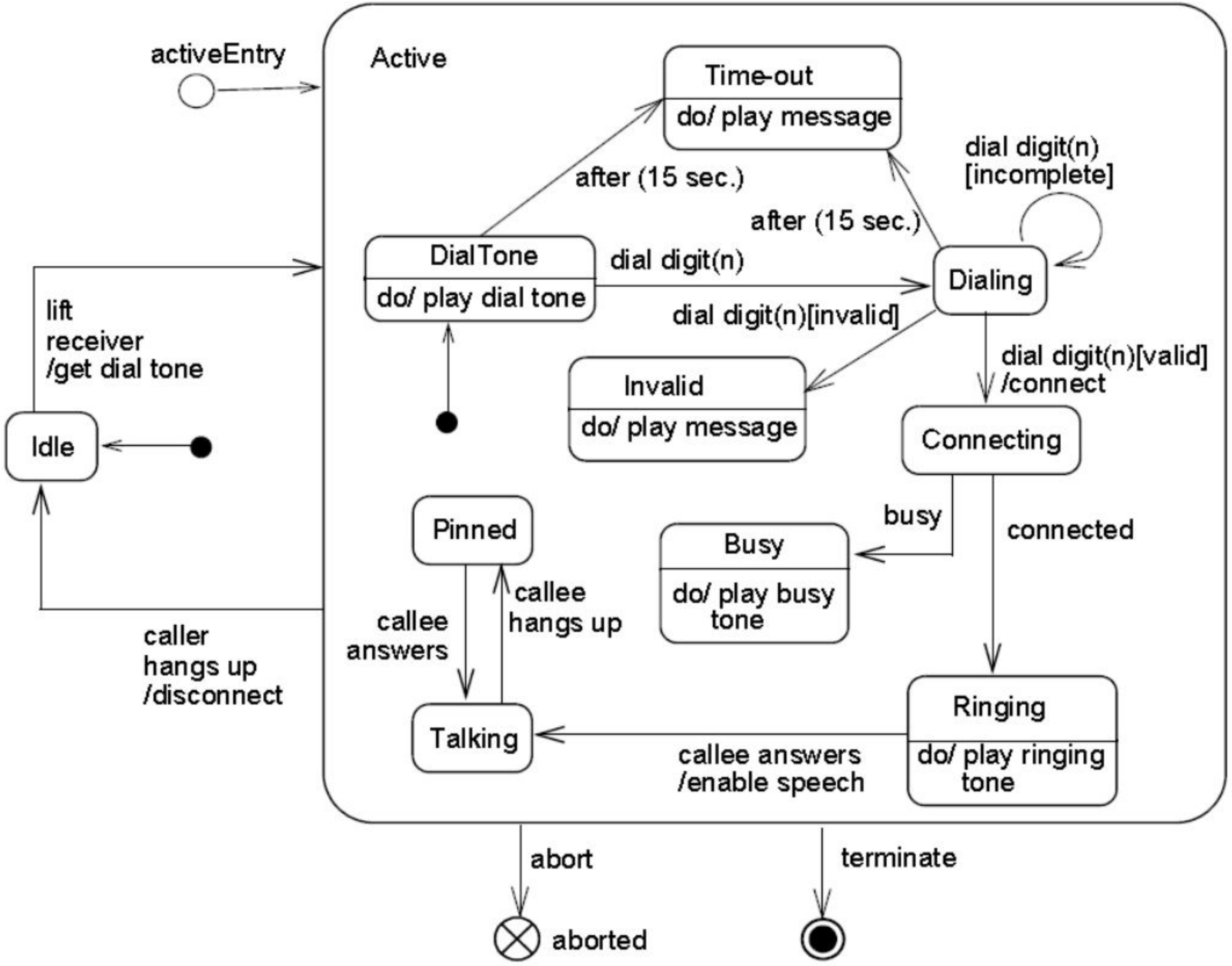
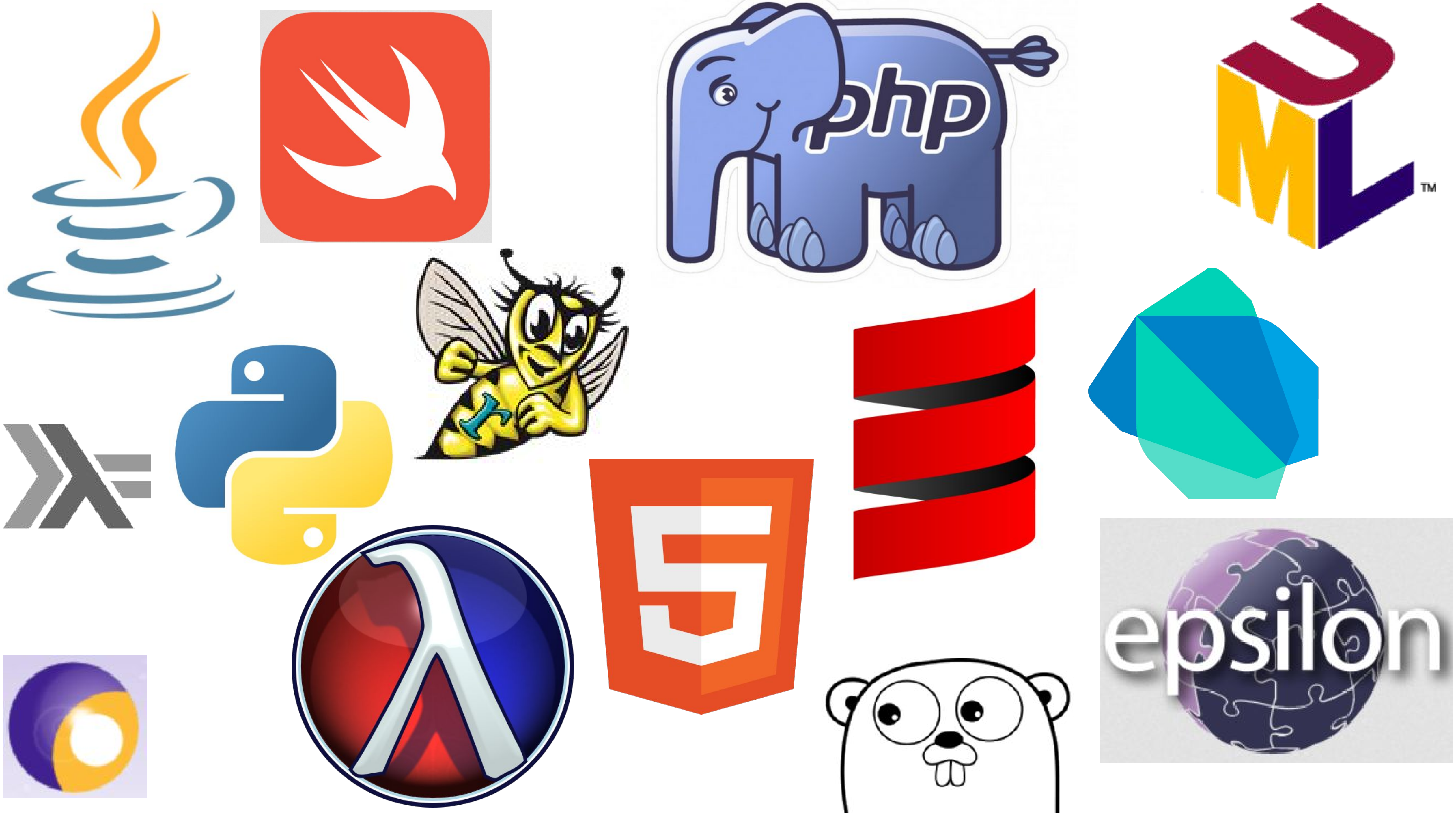
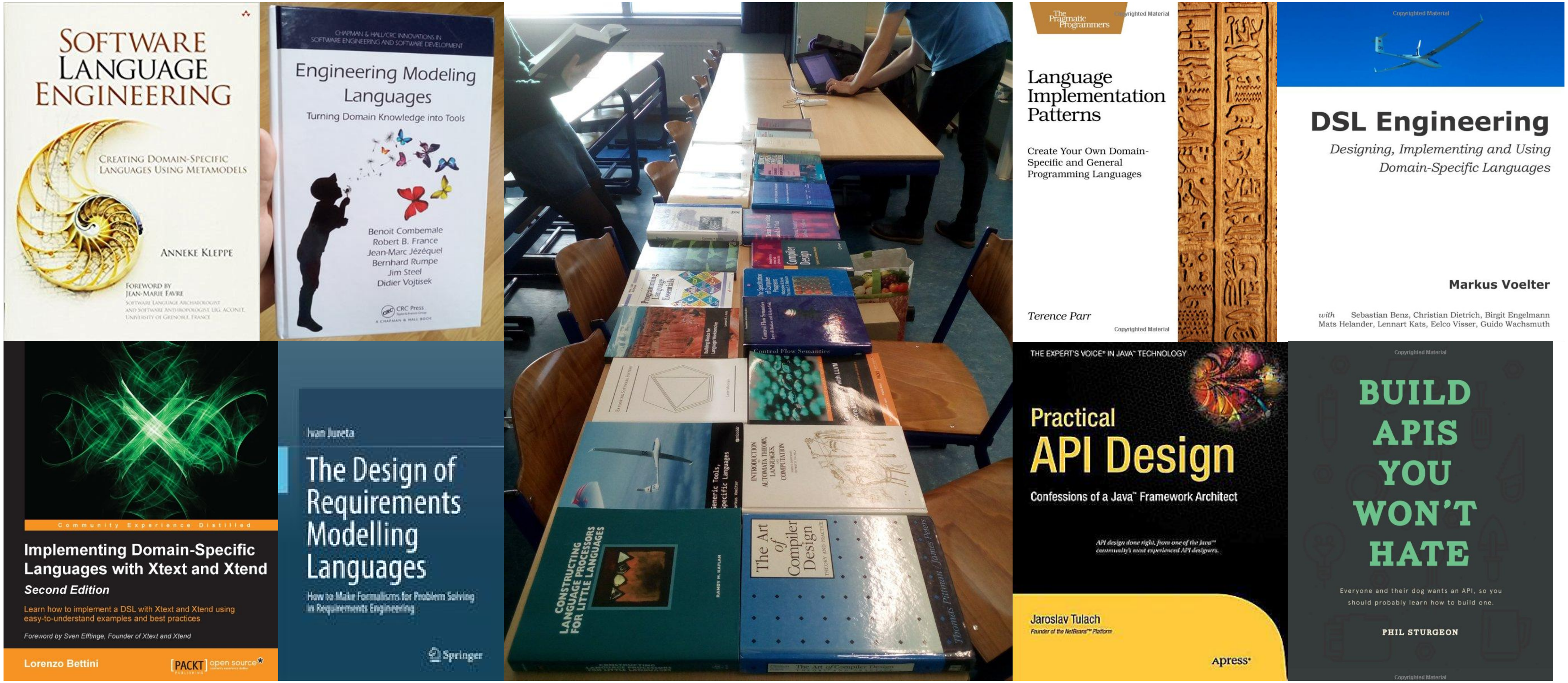

O.v W_v
i
1
I
$(\alpha+\kappa)_0$ to A
2
#
$\alpha+\kappa \to u$
3
O.v W_v'' for $\alpha+\kappa \leqq v < u$
W_v otherwise
A u_0
3
II
W_u to B
XI
$(u+1)_0$ to A
7
#
$u+1 \to u$
3
4
B W_u
III
$W_u^I - 2^{-12}\alpha$
+
4
−
IV
$W_u^I - 2^{-12}(\alpha+l)$
−
4
+
4
V
$W_u' = W_u + 2^{-19}\kappa$ to B
5
#
B W_u'
$W_u \to W_u'$
5
6
VI
$W_u^{'II} - 2^{-12}\alpha$
+
5
−
VII
$W_u^{'II} - 2^{-12}(\alpha+l)$
−
5
+
5
VIII
$W_u'' = W_u' + 2^{-39}\kappa$ to O.u
IX
$W_u'' = W_u'$ to O.u
6
6
X
$u - \alpha - \kappa - h + 1$
−
+
e
6
O.v W_v'' for $\alpha+\kappa \leqq v < \alpha+\kappa+h$
W_v otherwise
O.v W_v'' for $\alpha+\kappa \leqq v \leqq u$
W_v otherwise
α, l, h, κ
u

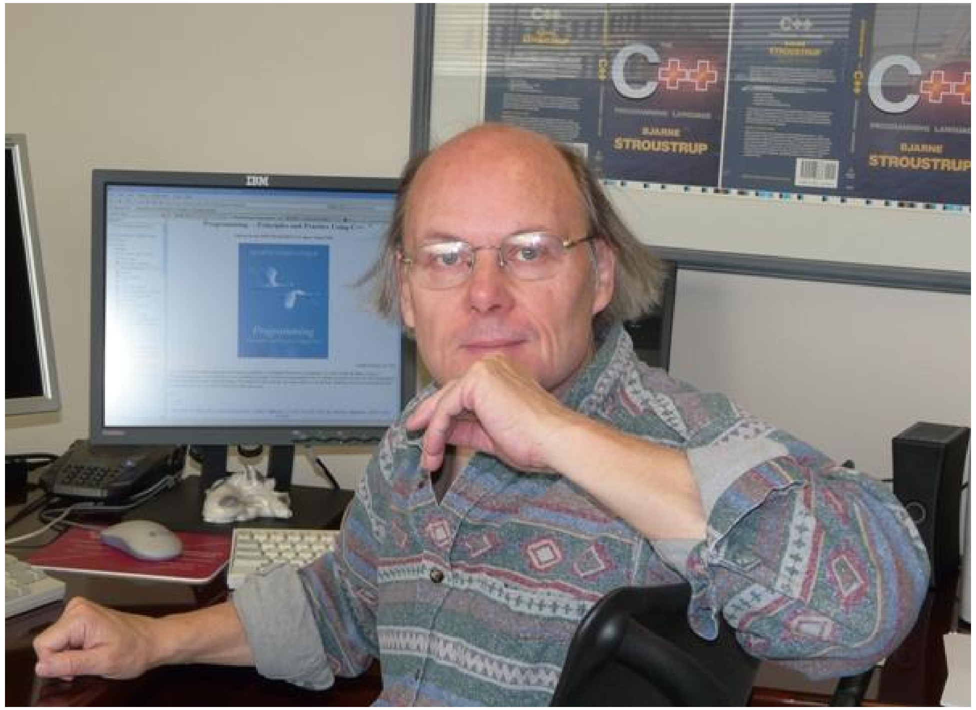
FIGURE 12.1


activeEntry
Active
Time-out
do/ play message
dial digit(n)
[incomplete]
after (15 sec.)
after (15 sec.)
DialTone
do/ play dial tone
dial digit(n)
Dialing
dial digit(n)[invalid]
lift
receiver
/get dial tone
Invalid
do/ play message
dial digit(n)[valid]
/connect
Idle
Connecting
busy
connected
Pinned
Busy
do/ play busy
tone
callee
hangs up
callee
answers
caller
hangs up
/disconnect
Talking
Ringing
do/ play ringing
tone
callee answers
/enable speech
abort
terminate
aborted


php
UML
™
5
epsilon

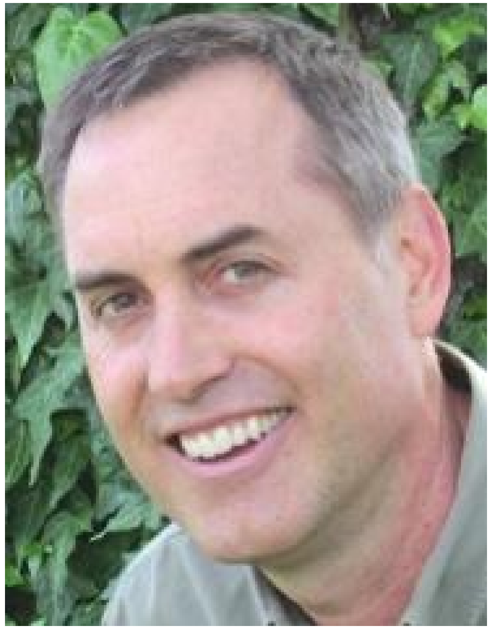
Where are we now?


SOFTWARE LANGUAGE ENGINEERING
CREATING DOMAIN-SPECIFIC LANGUAGES USING METAMODELS
ANNEKE KLEPPE
FOREWORD BY JEAN-MARIE FAVRE
CHAPMAN & HALL/CRC INNOVATIONS IN SOFTWARE ENGINEERING AND SOFTWARE DEVELOPMENT
Engineering Modeling Languages
Turning Domain Knowledge into Tools
Benoit Combemale
Robert B. France
Jean-Marc Jézéquel
Bernhard Rumpe
Jim Steel
Didier Vojtisek
CRC Press
The Pragmatic Programmers
Language Implementation Patterns
Create Your Own Domain-Specific and General Programming Languages
Terence Parr
DSL Engineering
Designing, Implementing and Using Domain-Specific Languages
Markus Voelter
with Sebastian Benz, Christian Dietrich, Birgit Engelmann Mats Helander, Lennart Kats, Eelco Visser, Guido Wachsmuth
Community Experience Distilled
Implementing Domain-Specific Languages with Xtext and Xtend
Second Edition
Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices
Foreword by Sven Efftinge, Founder of Xtext and Xtend
Lorenzo Bettini
[PACKT] open source
Ivan Jureta
The Design of Requirements Modelling Languages
How to Make Formalisms for Problem Solving in Requirements Engineering
Springer
Programming Language Essentials
Compiler Design
Control Flow Semantics
Generic Tools, Specific Languages
Introduction to Automata Theory, Languages, and Computation
Constructing Language Processors for Little Languages
The Art of Compiler Design
THE EXPERT'S VOICE® IN JAVA™ TECHNOLOGY
Practical API Design
Confessions of a Java™ Framework Architect
Jaroslav Tulach
Apress
BUILD APIS YOU WON'T HATE
Everyone and their dog wants an API, so you should probably learn how to build one.
PHIL STURGEON



“Most [...] language designers [...] impose their views on programmers and [...] some consider such imposition their duty”

[https://doi.org/10.1145/159544.159553]

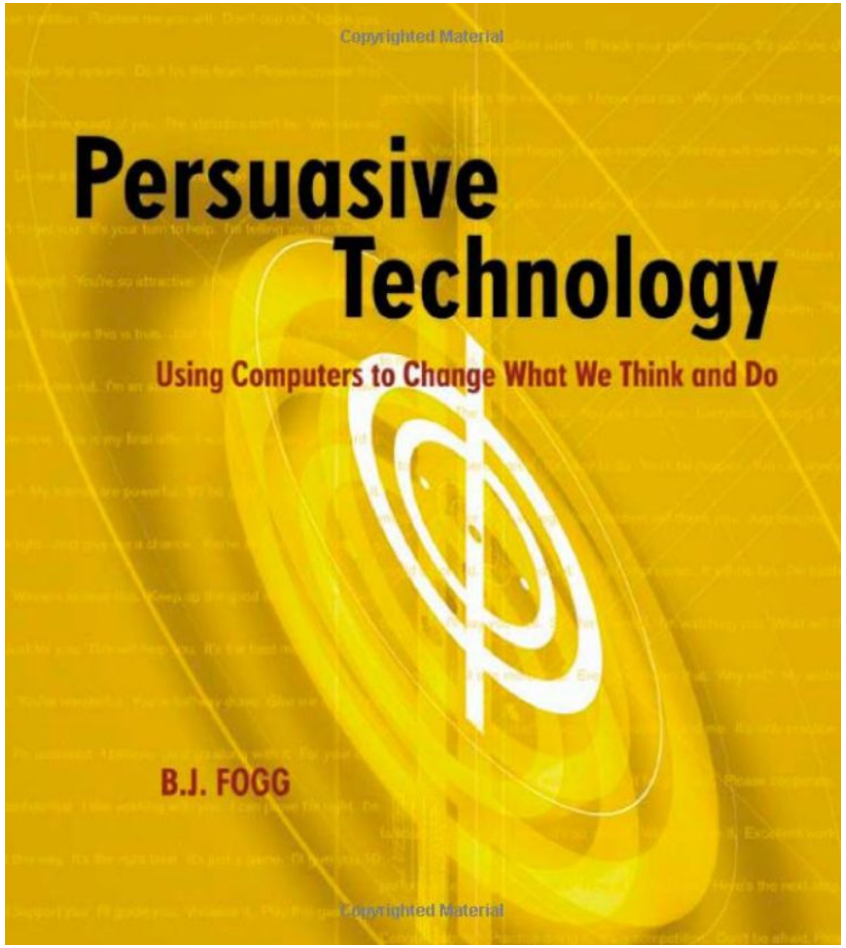


“Persuasive technology [...] is all about how to use computers—whether it's mobile phones, websites, video games—to change people’s attitudes and especially their behaviours”

[http://www.bjfogg.com]

Persuasive Technology

- Behaviour ::= Trigger Ability Motivation ;
- Reduction (persuade by simplification)
- Tunneling (by guiding)
- Tailoring (by customisation)
- Suggestion (by intervening)
- Surveillance (by observation)
- Conditioning (by training)



Design with Intent

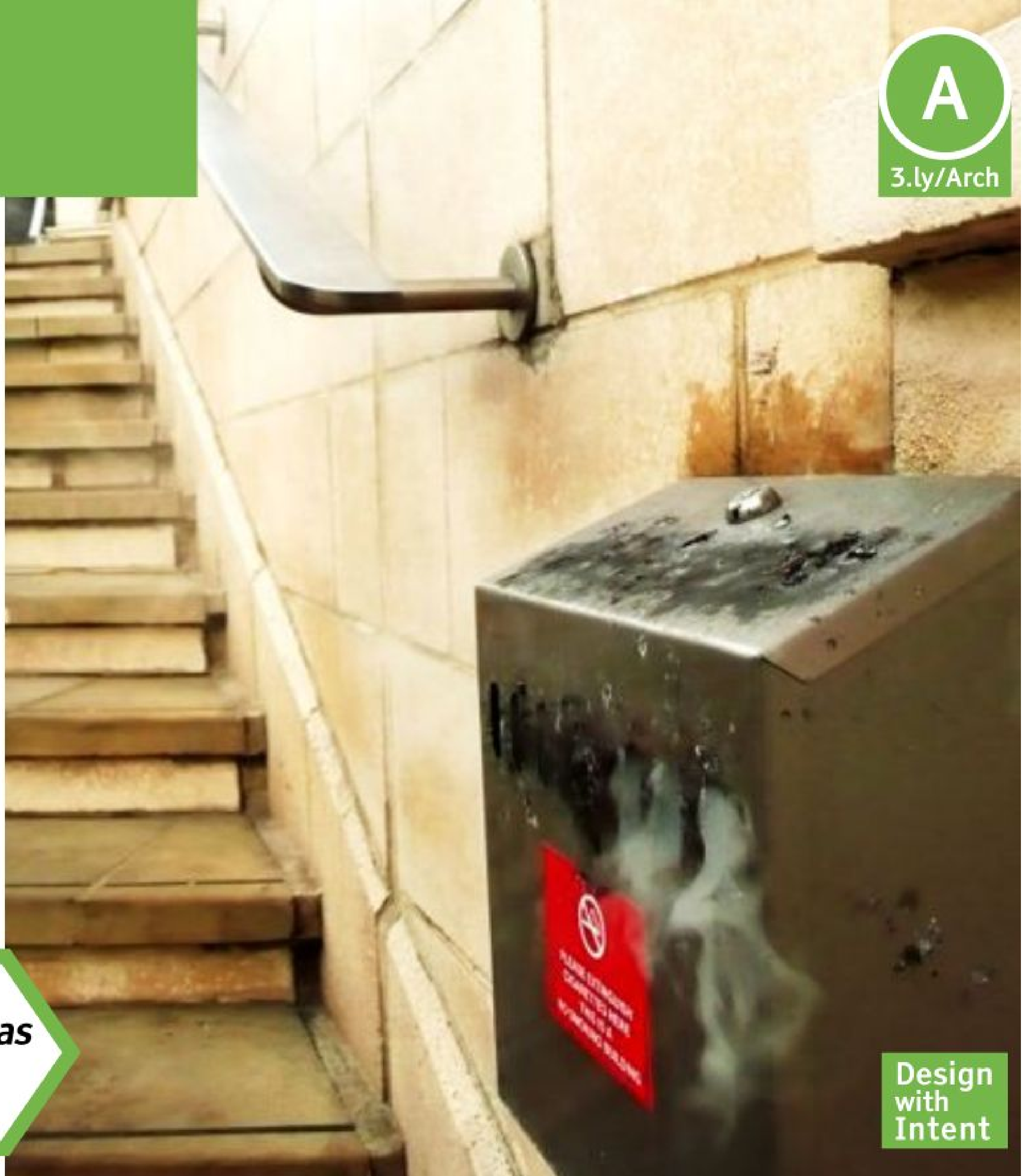
- The purpose of design is to change how users behave
- Apply techniques intentionally!
- Learn from analogous systems
- Naturalistic decision making
- Lateral thinking and divergent production

Angles

A

3.ly/Arch

Can you slant or angle things so some actions are easier than others?



Some cigarette bins are sold to authorities using the sloping top as a feature, discouraging people leaving litter on top

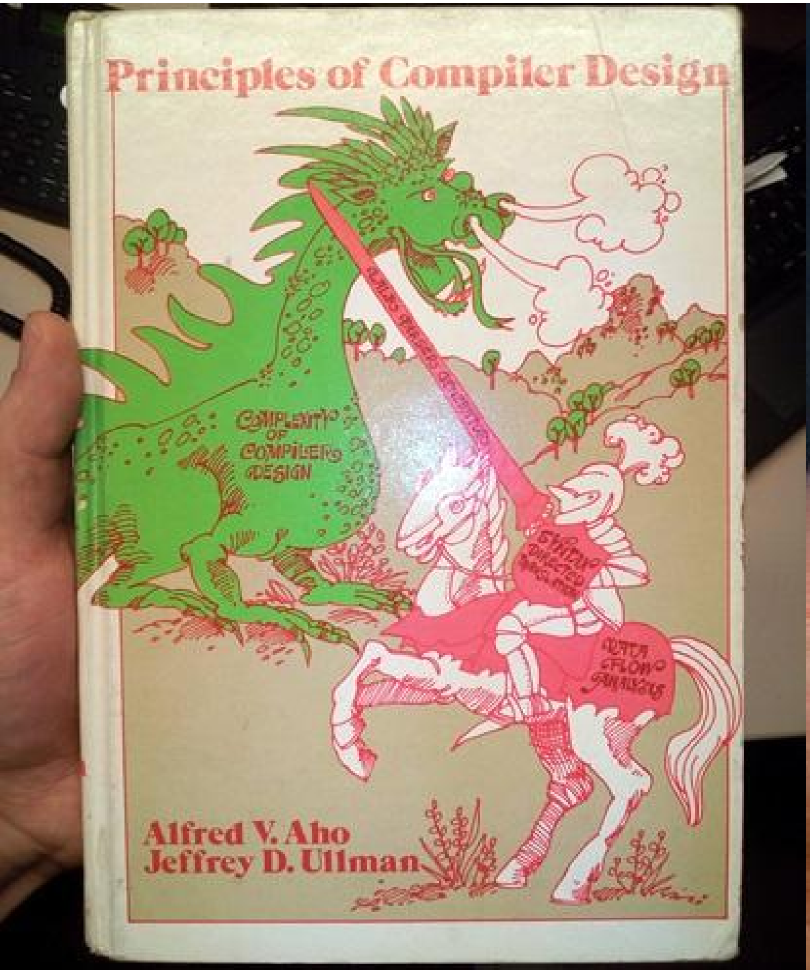
Access Modifier

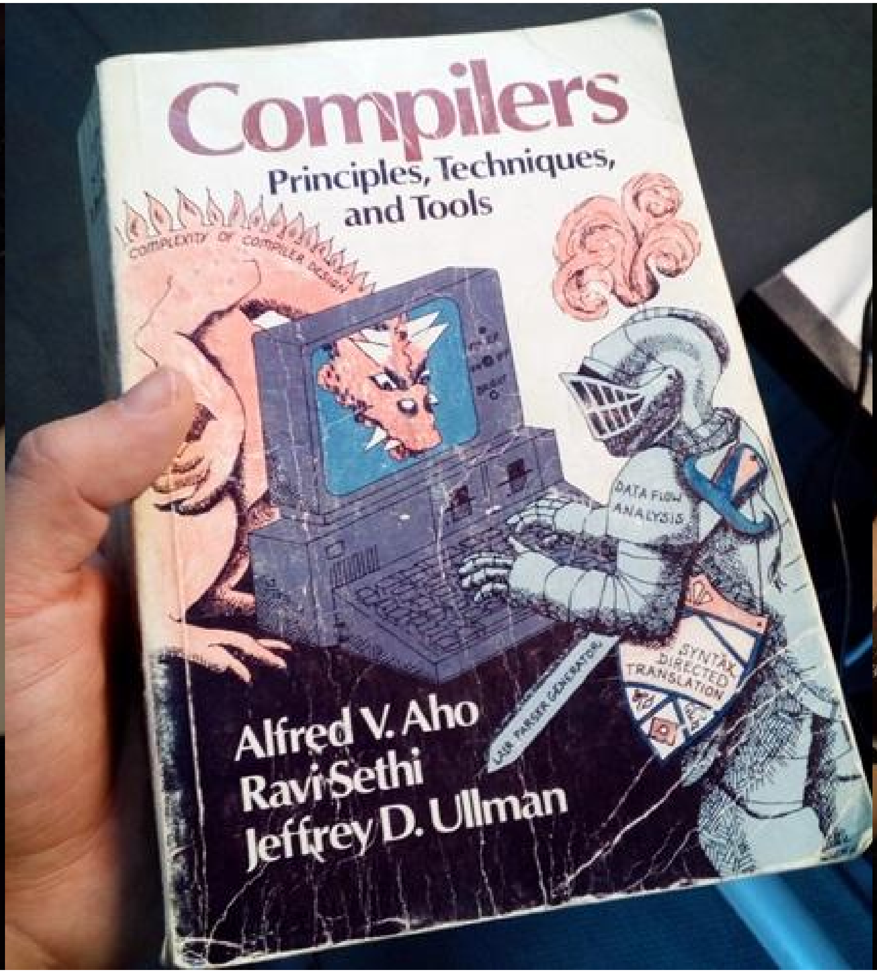
Annotate components with information about how others are allowed or not allowed to access them. Access can be limited by inheritance (*protected* in C++), modular structure (*internal* in C#), etc. The most popular modifiers are *public* (everyone welcome) and *private* (fully restricted). Similar modifiers can be used to manage scope, such as *global* and *nonlocal* in Python.

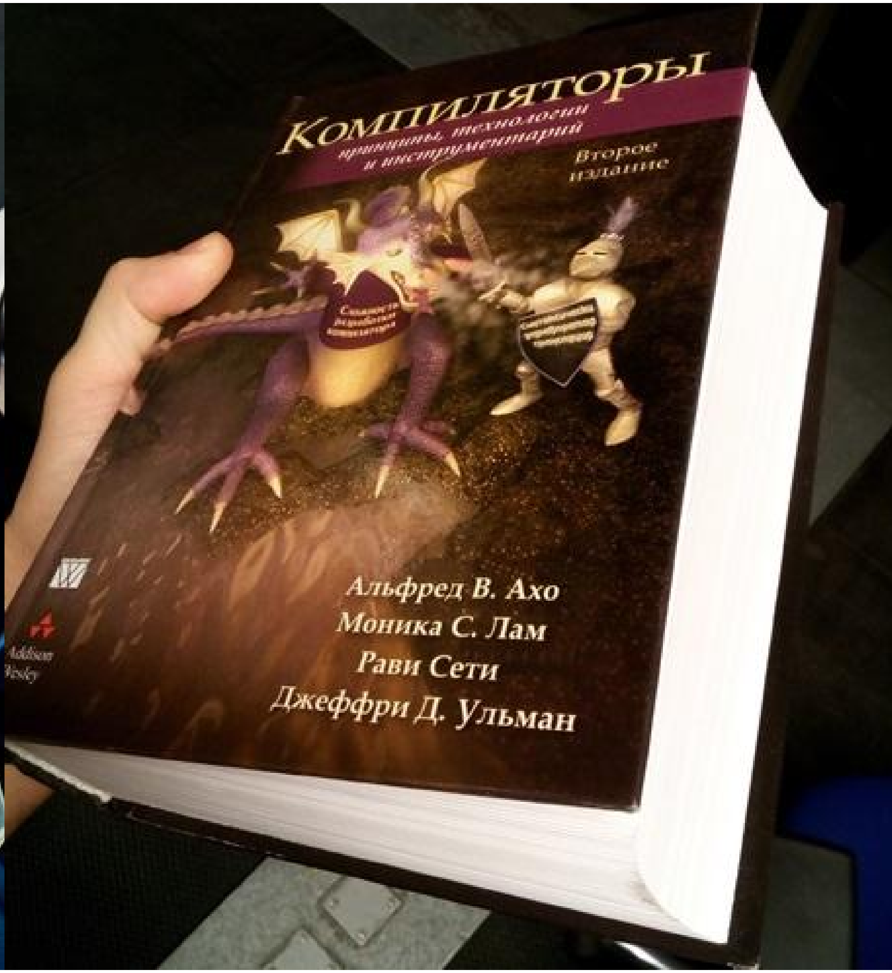
Dwl:Angles

coding
sampling
sorting
memoing
theorising

Dragon Books


Principles of Compiler Design
COMPLEXITY OF COMPILER DESIGN
SYNTAX DIRECTED TRANSLATION
DATA FLOW ANALYSIS
Alfred V. Aho
Jeffrey D. Ullman


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION
LALR PARSER GENERATOR
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

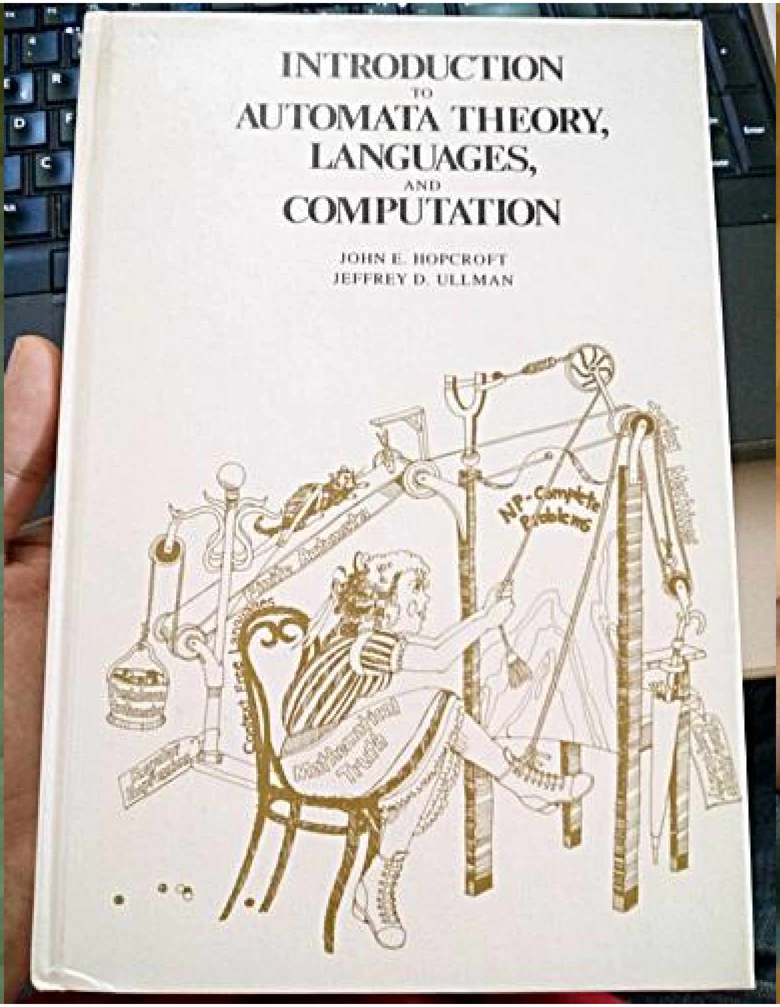


Parsing Techniques

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

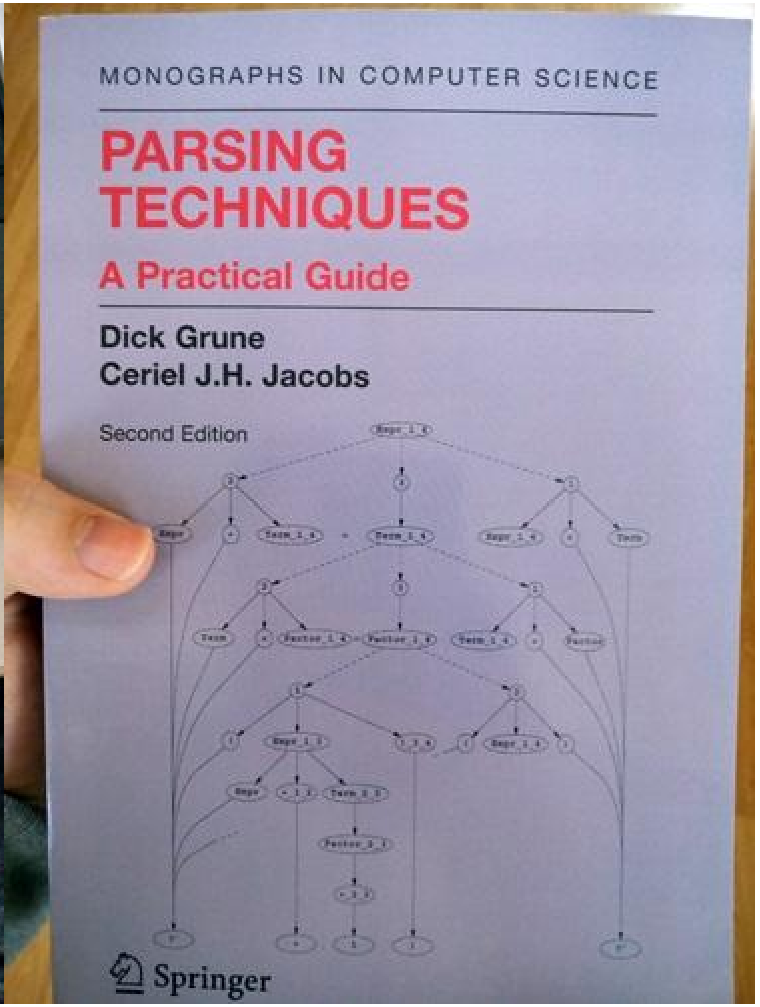
А.Оллонгрен

ОПРЕДЕЛЕНИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ИНТЕРПРЕТИРУЮЩИМИ АВТОМАТАМИ



INTRODUCTION TO AUTOMATA THEORY, LANGUAGES, AND COMPUTATION

JOHN E. HOPCROFT
JEFFREY D. ULLMAN



MONOGRAPHS IN COMPUTER SCIENCE

PARSING TECHNIQUES

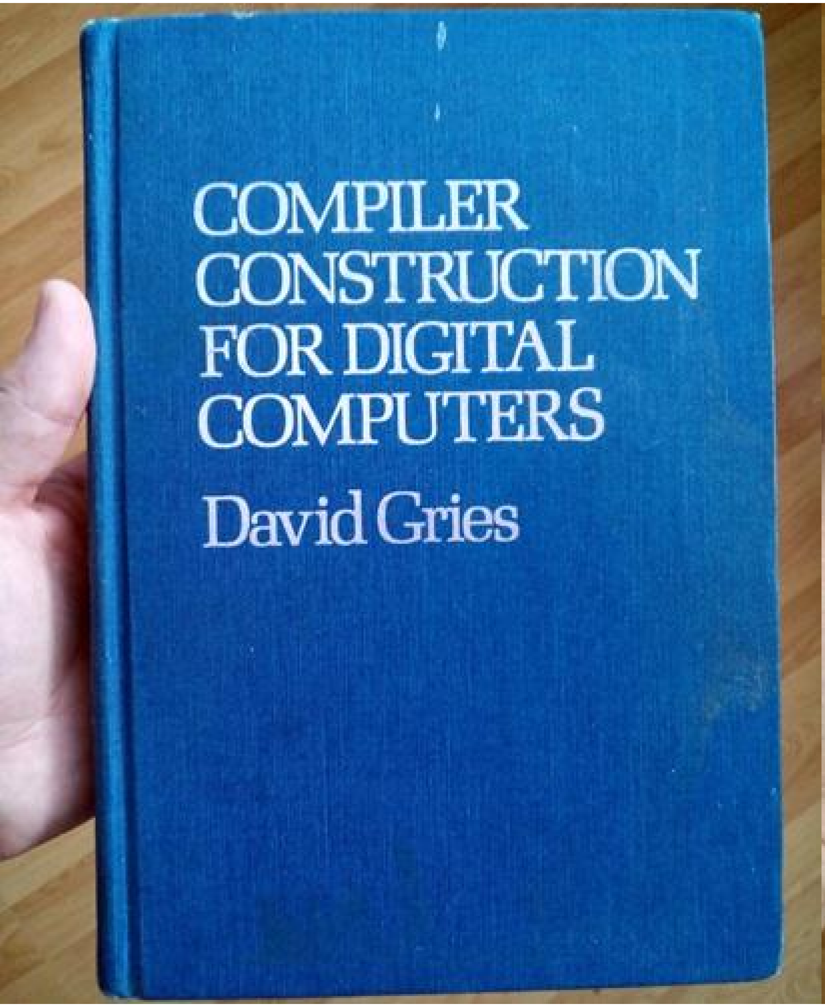
A Practical Guide

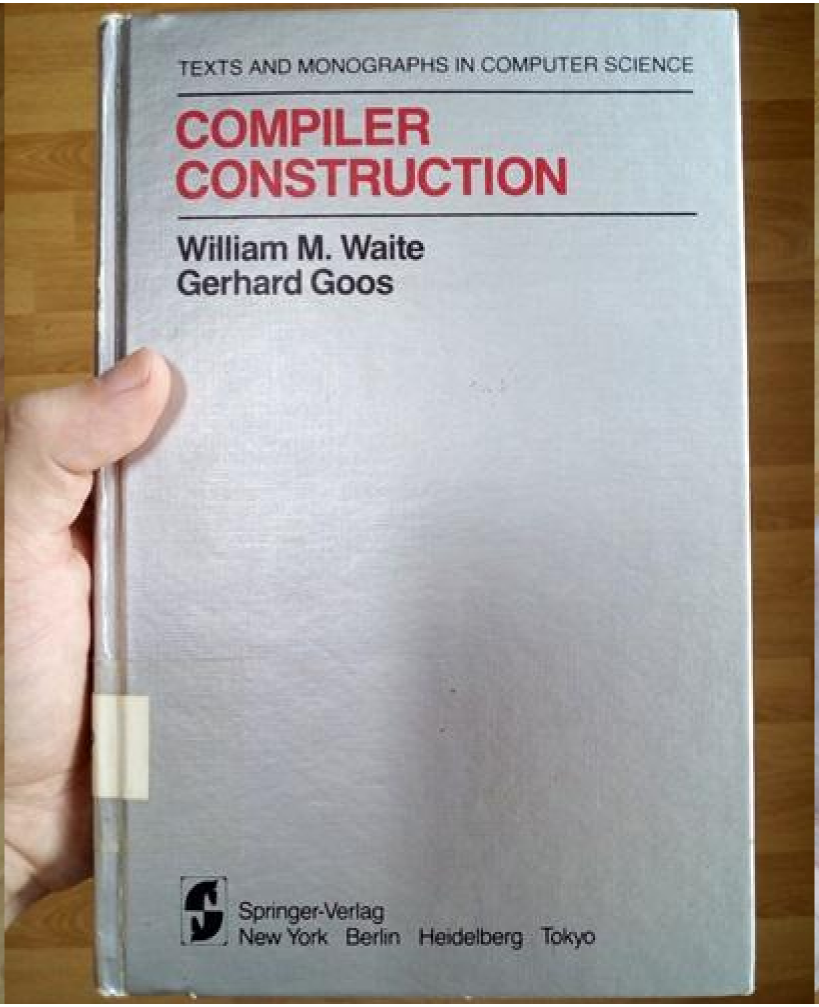
Dick Grune
Ceriel J.H. Jacobs

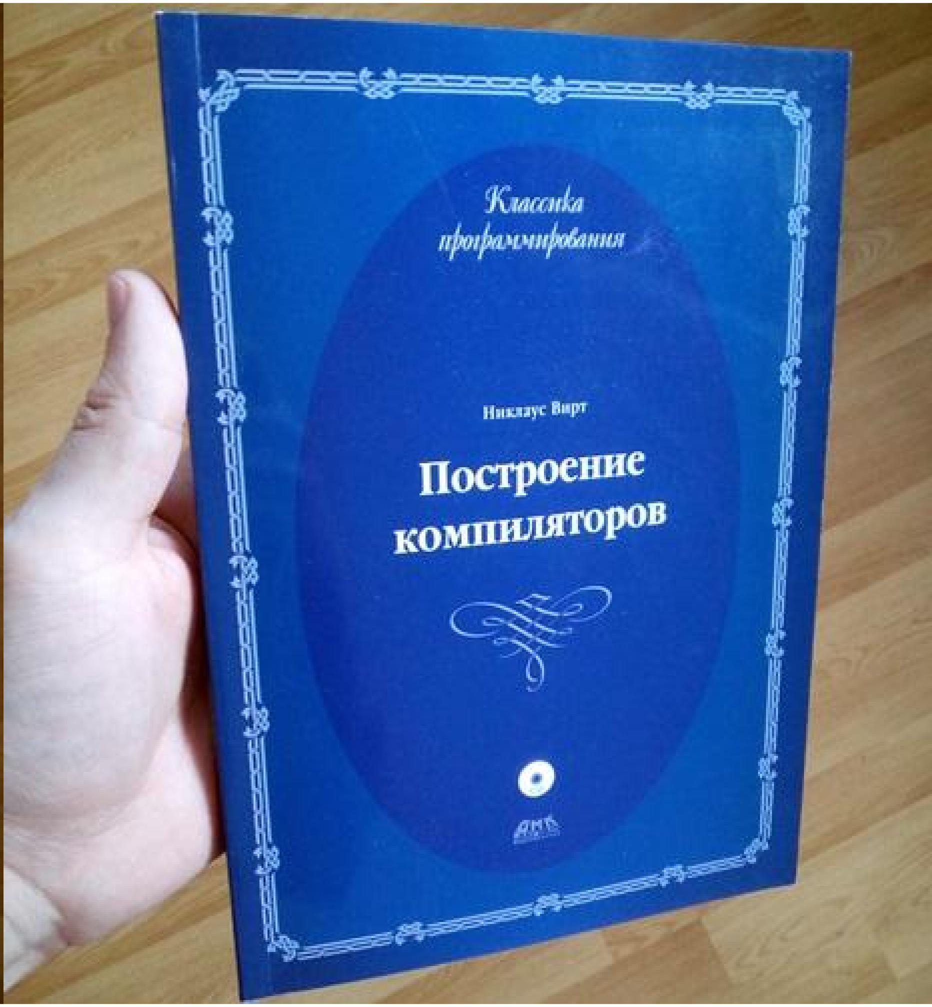
Second Edition

Springer

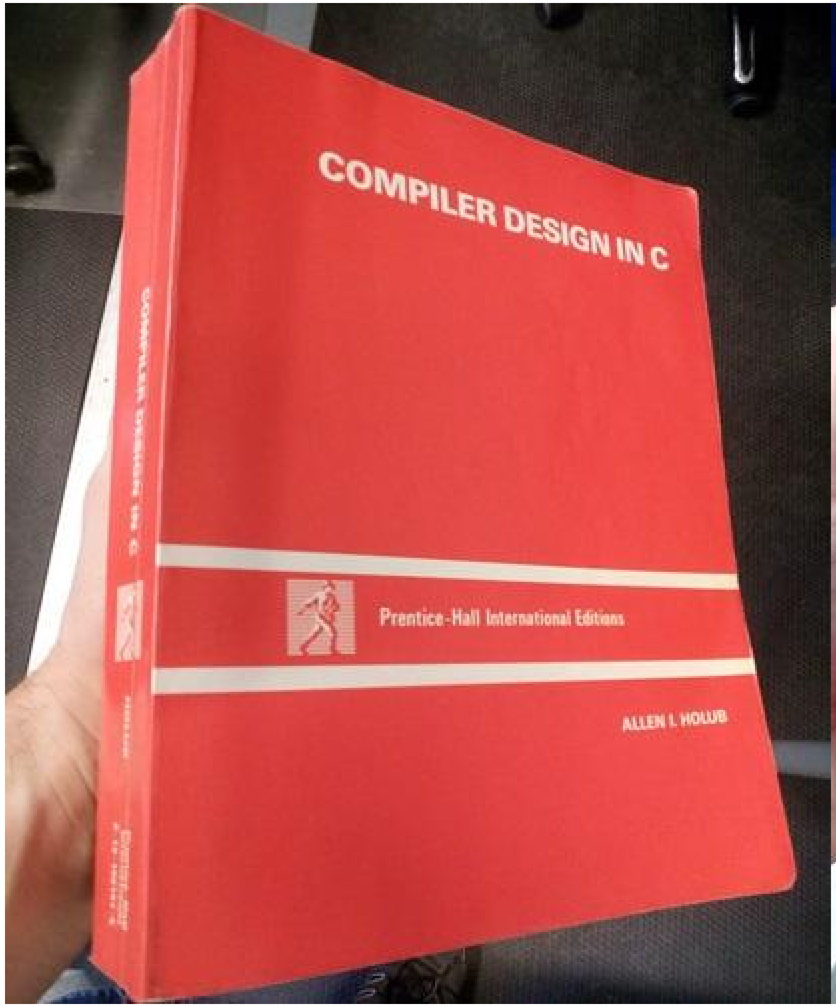
Compiler Construction

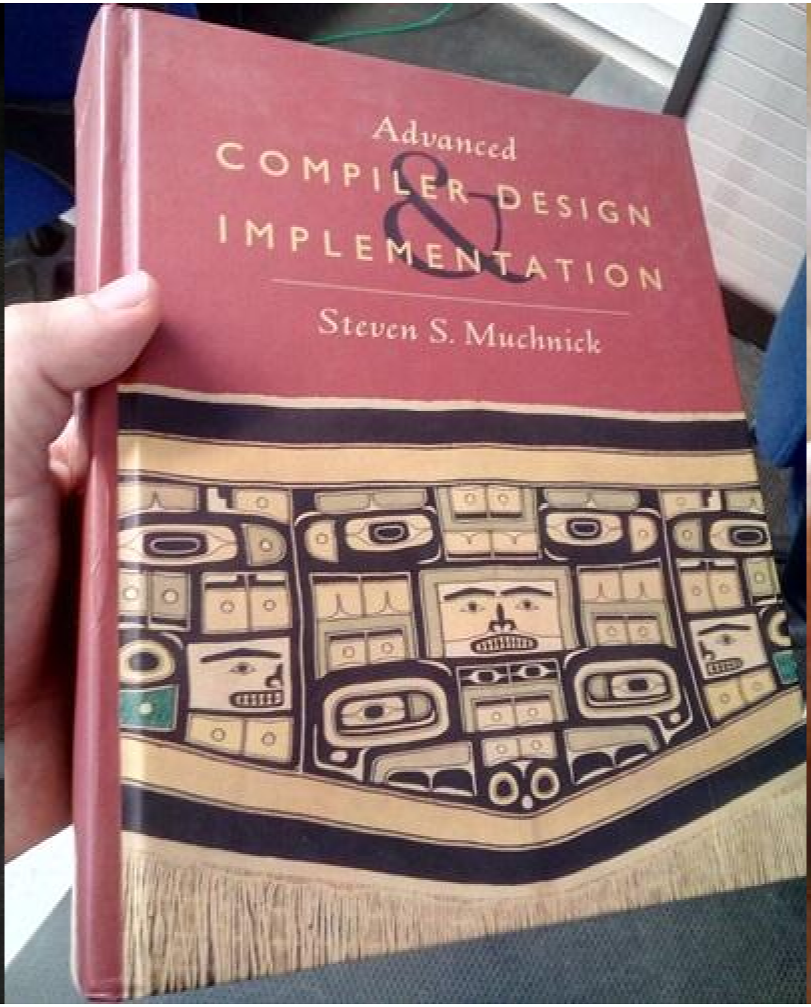


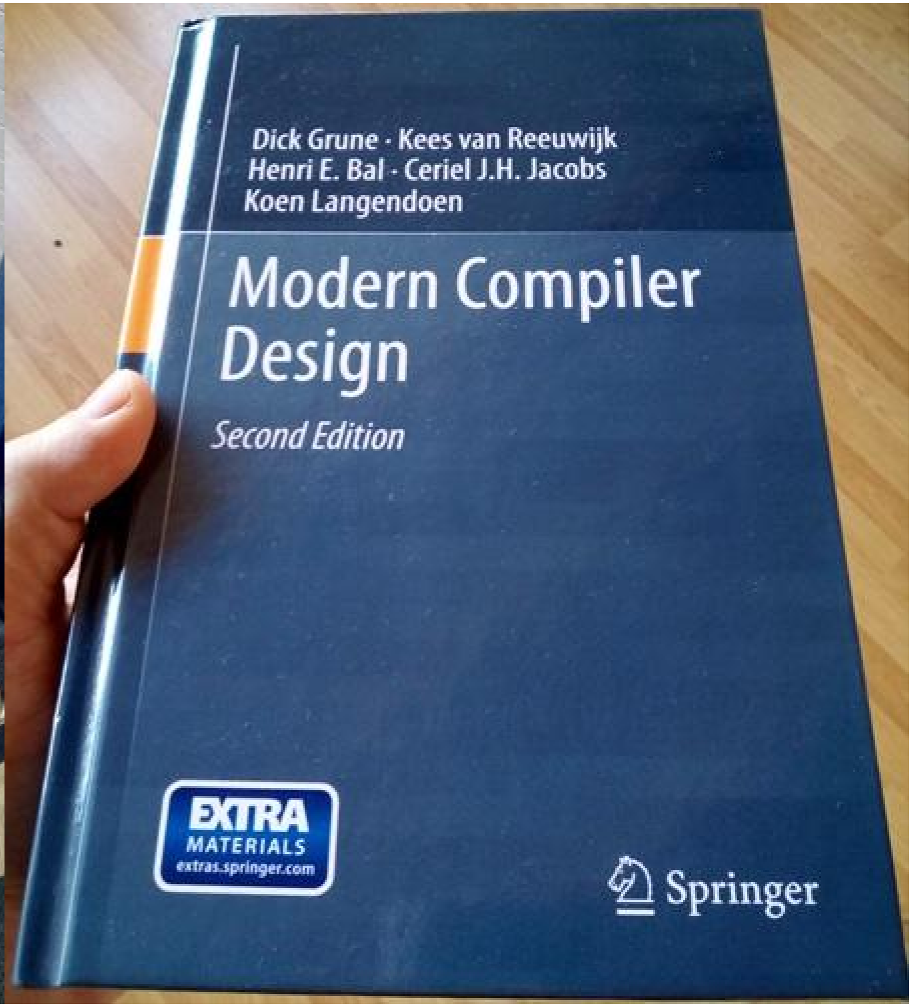

TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE
COMPILER
CONSTRUCTION
William M. Waite
Gerhard Goos
Springer-Verlag
New York Berlin Heidelberg Tokyo



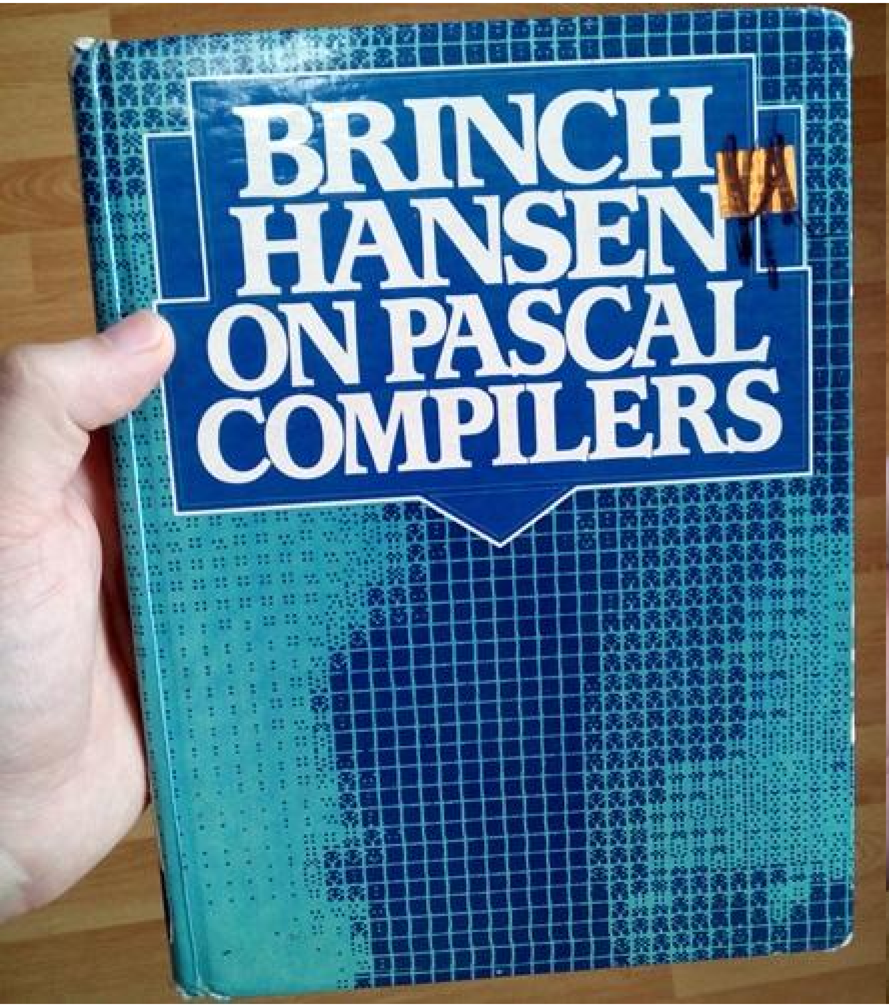
Compiler Design

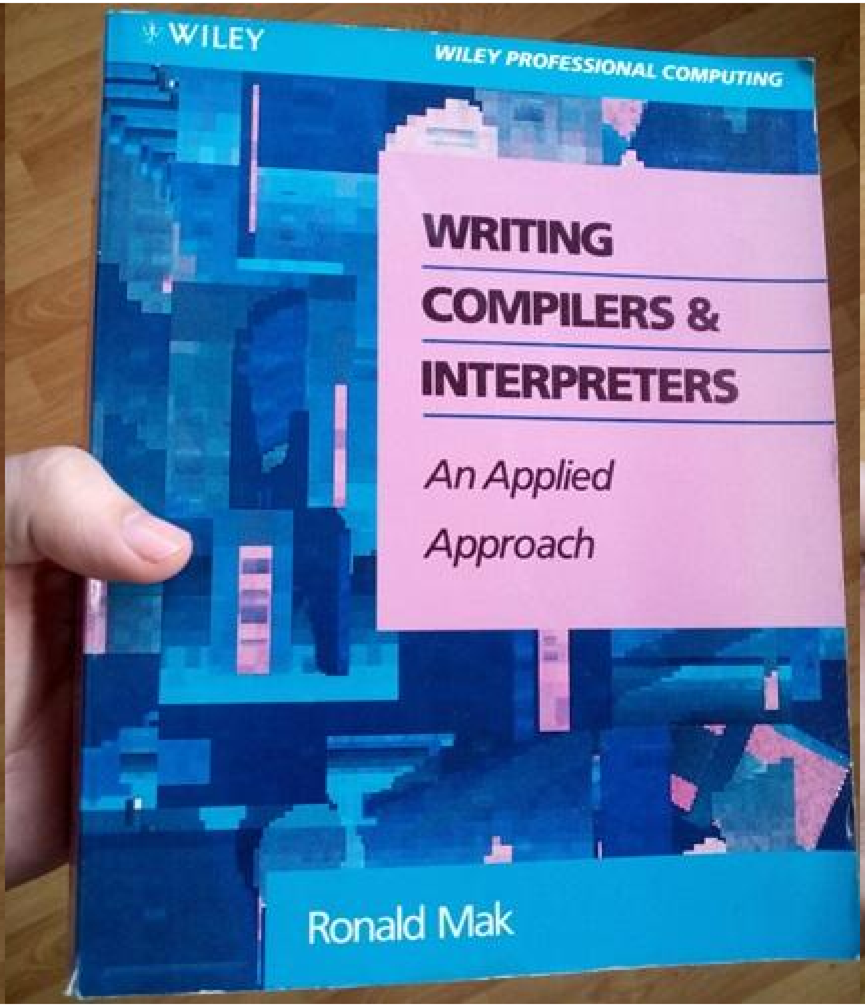


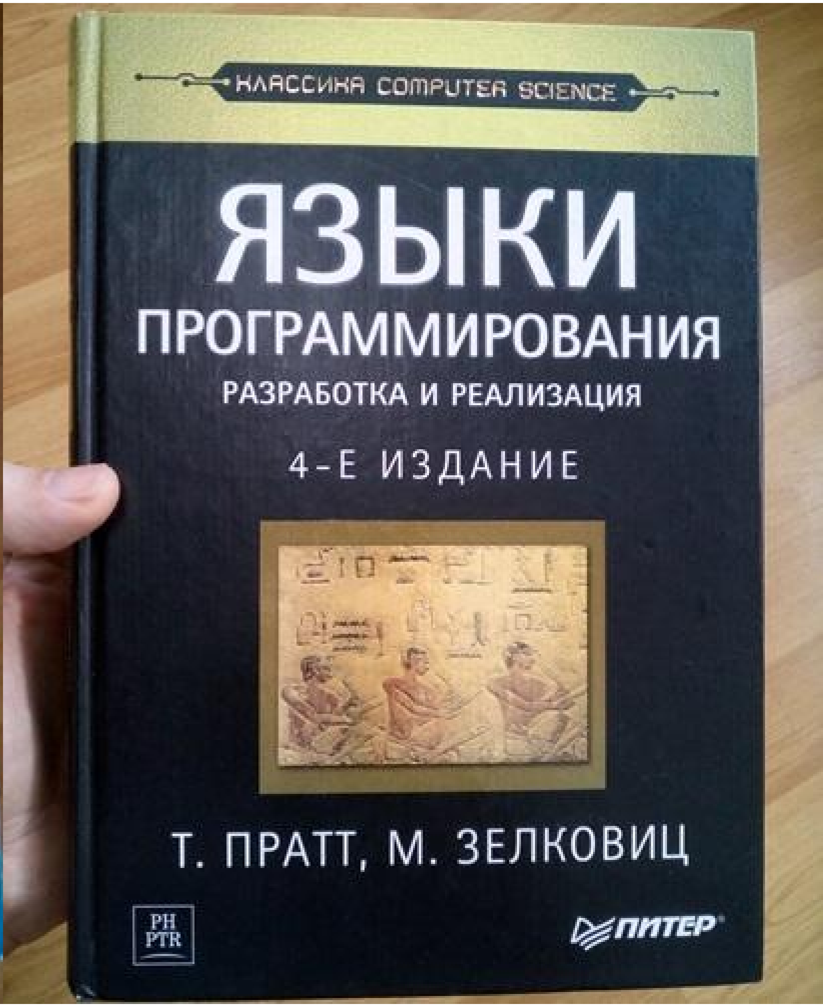



Dick Grune · Kees van Reeuwijk
Henri E. Bal · Ceriel J.H. Jacobs
Koen Langendoen
Modern Compiler Design
Second Edition
EXTRA MATERIALS extras.springer.com
Springer

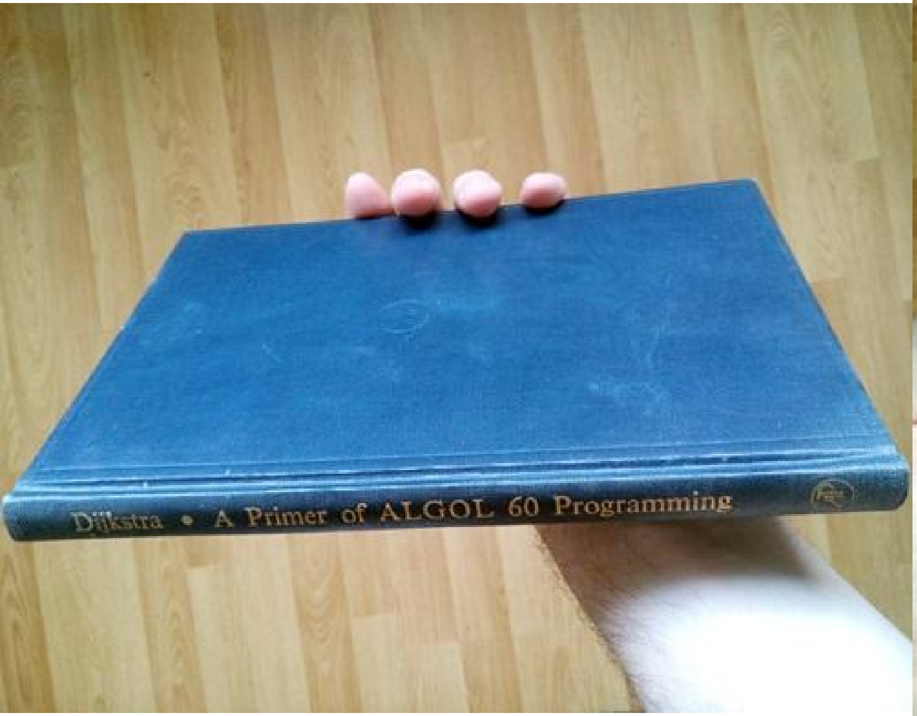
Language Implementation

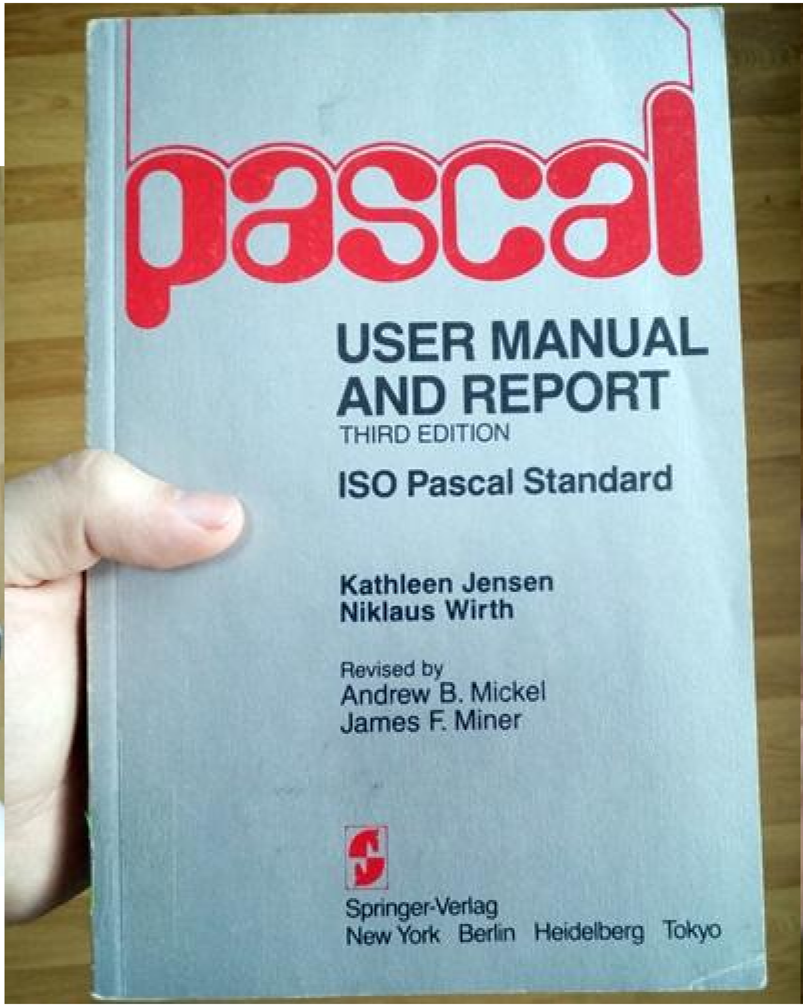


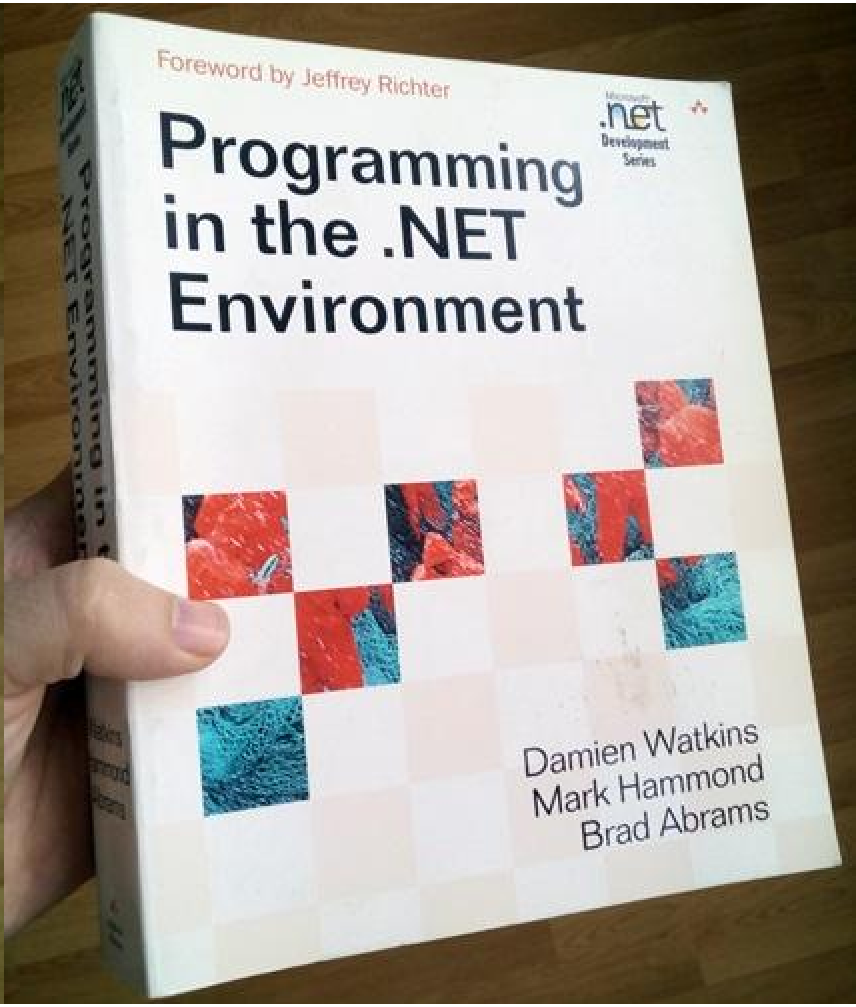

WILEY
WILEY PROFESSIONAL COMPUTING
WRITING COMPILERS & INTERPRETERS
An Applied Approach
Ronald Mak



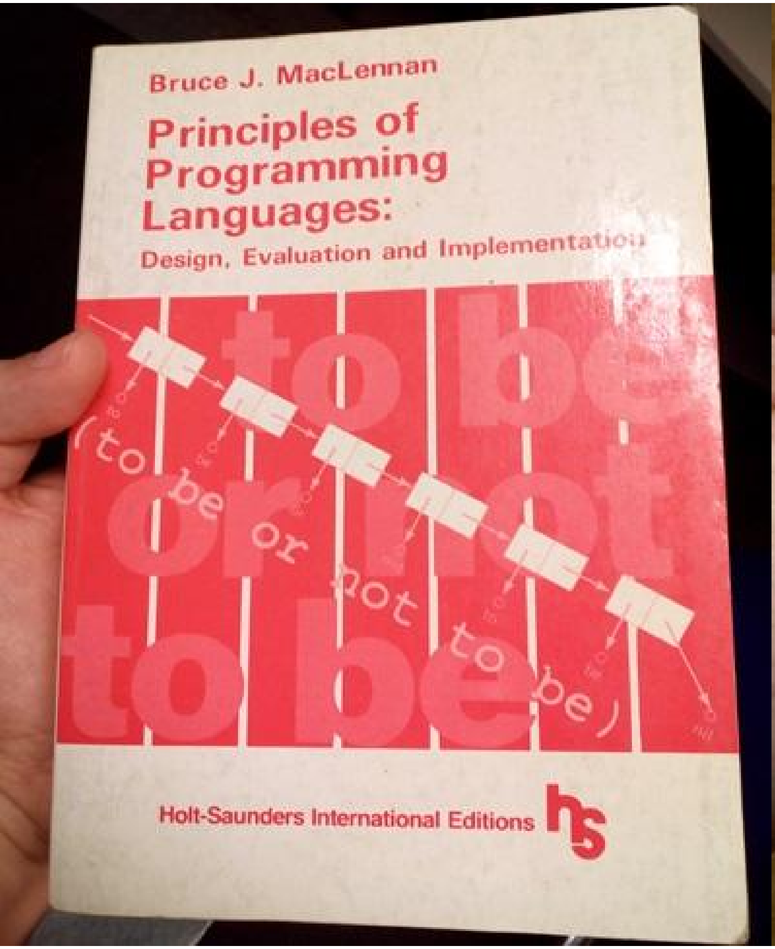
Language Documentation

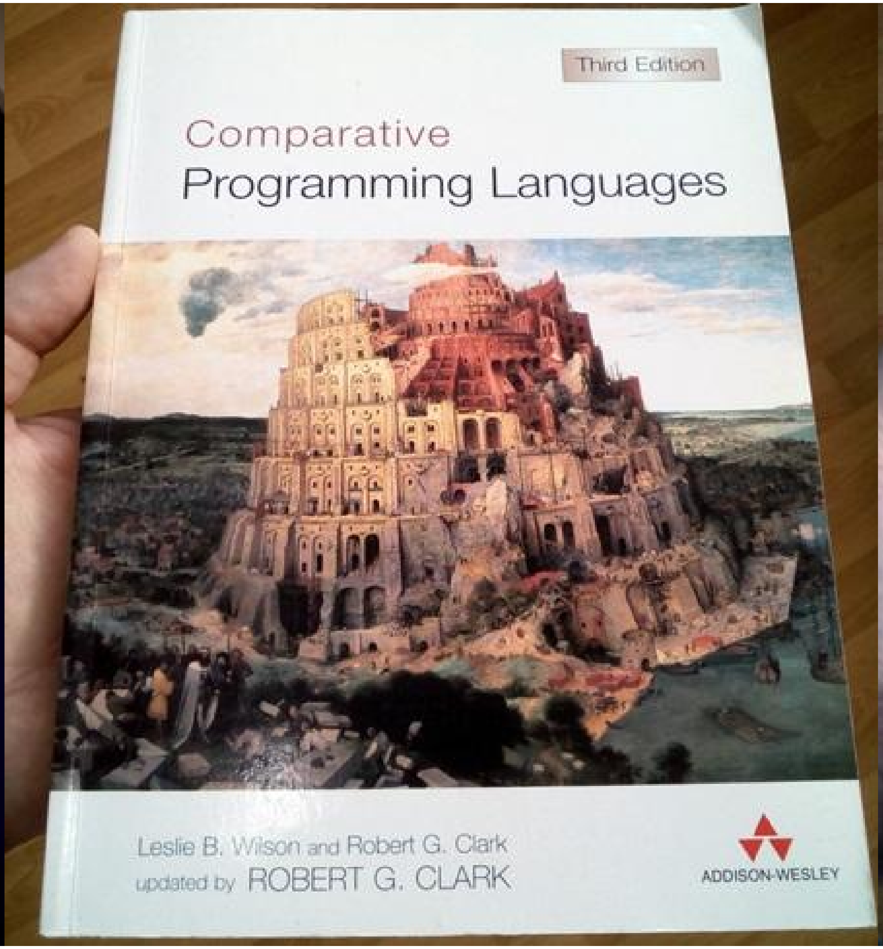


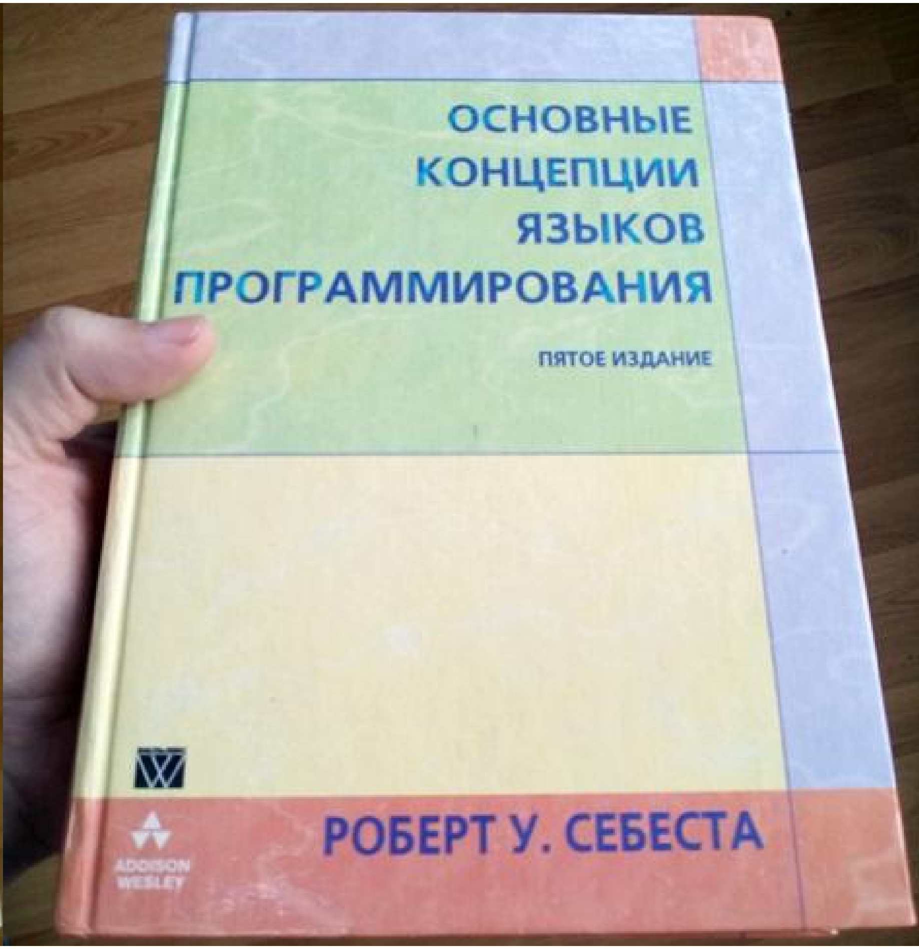



Foreword by Jeffrey Richter
Programming in the .NET Environment
.net Development Series
Damien Watkins
Mark Hammond
Brad Abrams

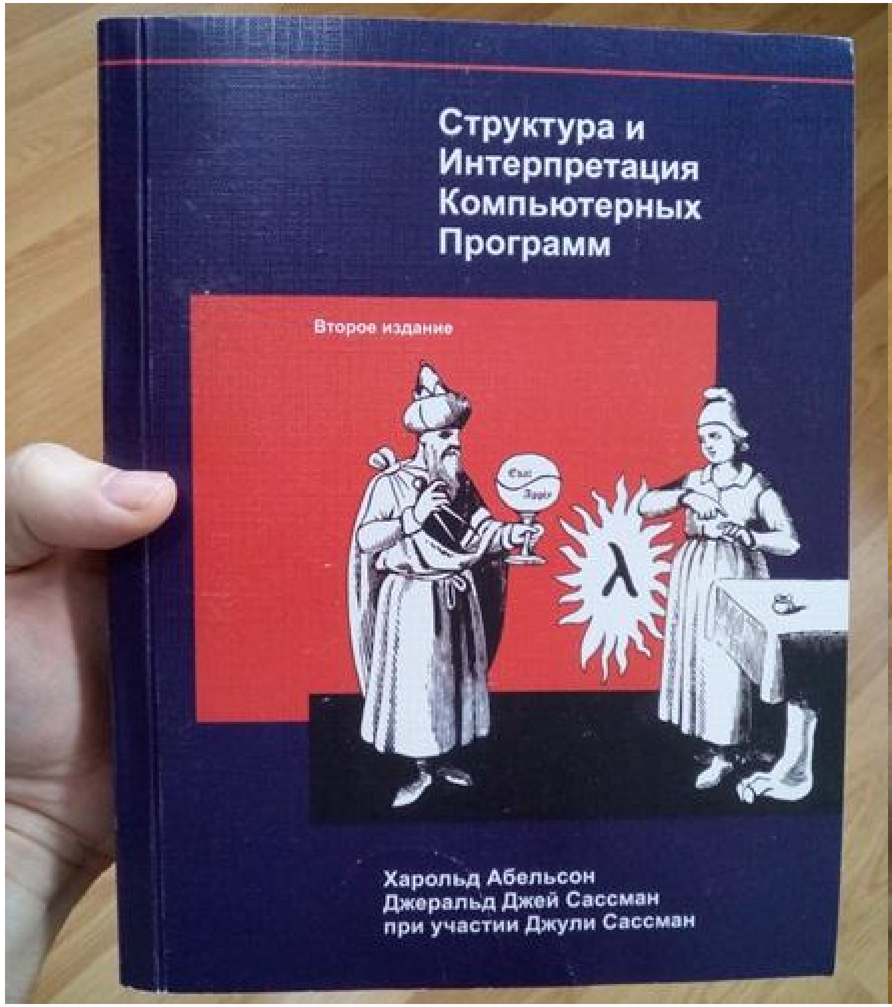
Programming Languages

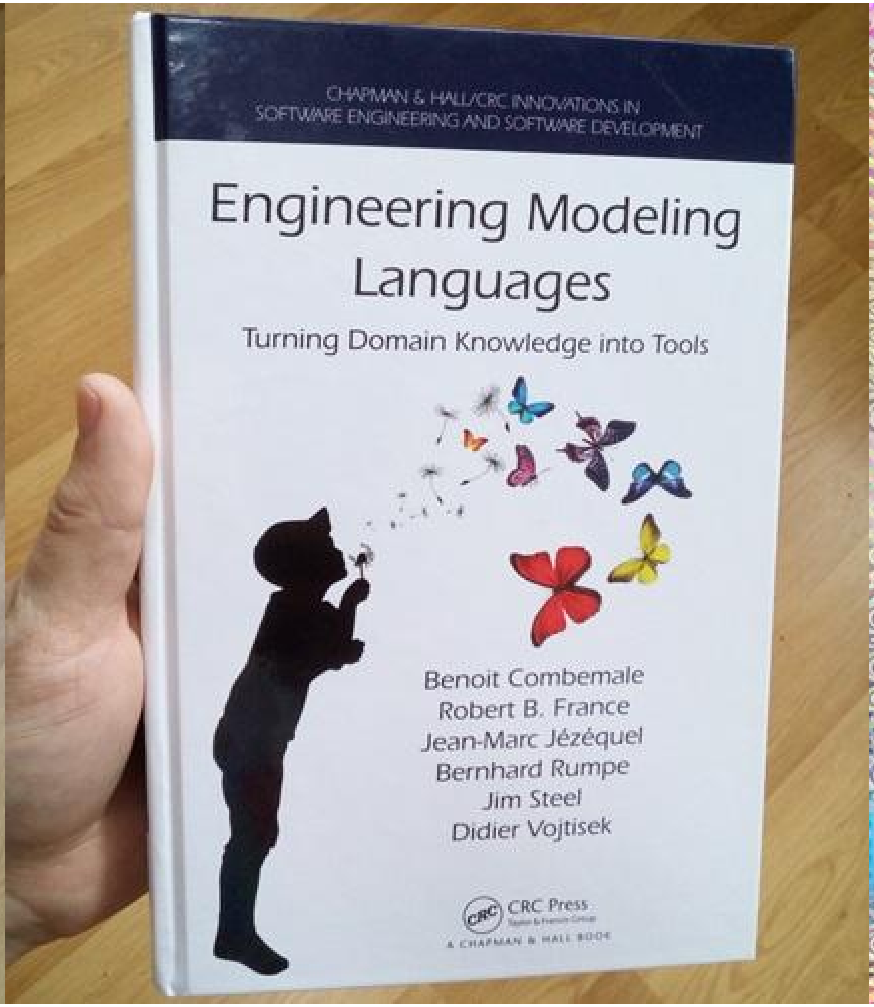

Bruce J. MacLennan
Principles of Programming Languages:
Design, Evaluation and Implementation
to be or not to be
(to be or not to be)
Holt-Saunders International Editions
hs

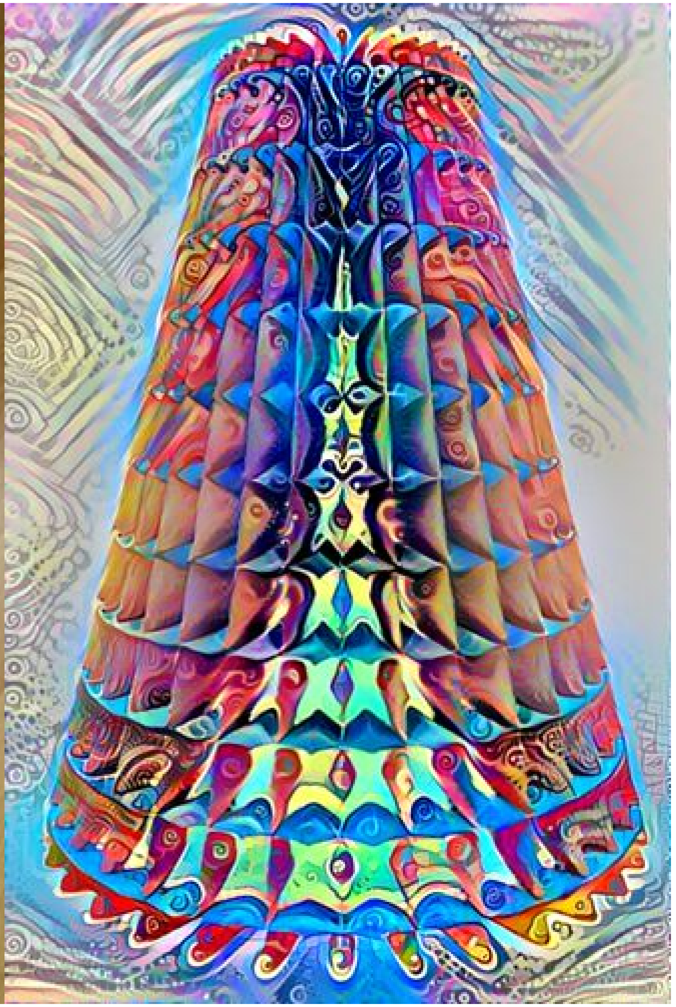

Third Edition
Comparative
Programming Languages
Leslie B. Wilson and Robert G. Clark
updated by ROBERT G. CLARK
ADDISON-WESLEY



Software Languages


Структура и Интерпретация Компьютерных Программ
Второе издание
λ
Харольд Абельсон
Джеральд Джей Сассман
при участии Джули Сассман


CHAPMAN & HALL/CRC INNOVATIONS IN SOFTWARE ENGINEERING AND SOFTWARE DEVELOPMENT
Engineering Modeling Languages
Turning Domain Knowledge into Tools
Benoit Combemale
Robert B. France
Jean-Marc Jézéquel
Bernhard Rumpe
Jim Steel
Didier Vojtisek
CRC Press
A CHAPMAN & HALL BOOK



Access Modifier

Annotate components with information about how others are allowed or not allowed to access them. Access can be limited by inheritance (*protected* in C++), modular structure (*internal* in C#), etc. The most popular modifiers are *public* (everyone welcome) and *private* (fully restricted). Similar modifiers can be used to manage scope, such as *global* and *nonlocal* in Python.

Dwl:Angles, DB-PD:65, CD-AH:42, LD-WH:58

Encapsulation

Most high level language abstract from low level details like video memory access, memory allocation, register values, caching, etc. Depending on the language design and philosophy, these features may be prohibited or just hard to find for beginners. Data structures can also be encapsulated by bundling them into records or classes, and code can be organised in hierarchical modules and subprograms.

Dwl:Hiding things, LI-PZ:236, PL-RS:37, PL-WC:104, PL-BM:12, LD-WH:229, SL-AS:15, SL-RL:19

Alphabet §

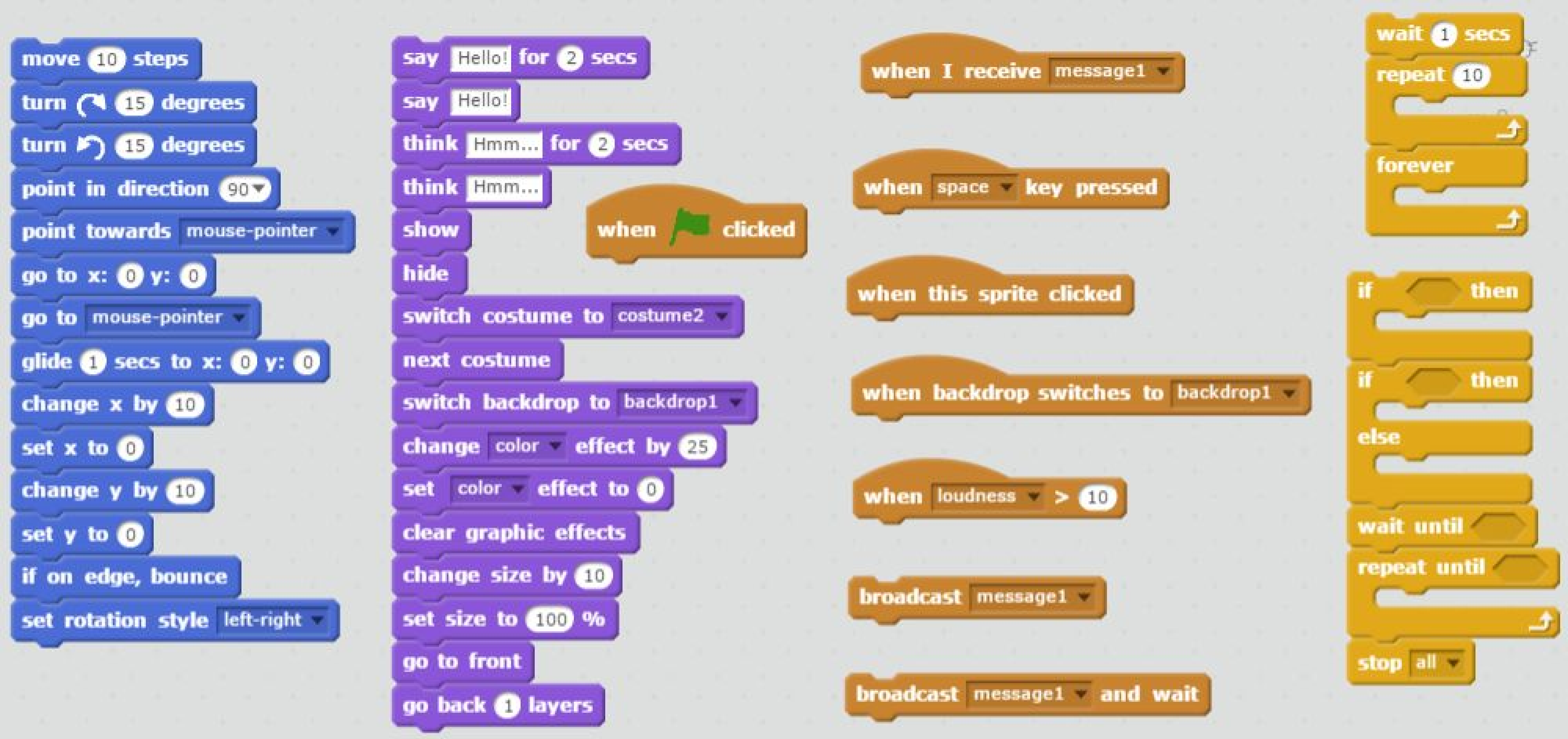
The basic alphabet is often taken for granted, especially for textual languages, but it is an important design aspect. In some languages (APL being the extreme) the alphabet is extremely broad, with specific symbols being used for built-in operators, which shifts the visual feel of the language closer to mathematics. In other languages keywords are taken from English, which limits language appeal to some groups of users (and may lead to reimplementations with translated keywords).

Dwl:Perceived affordances, **DB-GD:28**, **DB-RD:92**, **DB-PD:165**, **CC-DG:15**, **CC-NW:10**, **CD-AH:52**, **LI-BH:10**, **PT-AO:34**, **PT-HU:1**, **PT-GJ:6**, **LD-ED:5**

USASCII code chart

b_7 b_6 b_5 → Bits					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ZENITH
DATA SYSTEMS
Z-STATION EL
PHILIPS
Esc
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
Tab
Caps Lock
Shift
Ctrl
Meta
Backspace
Enter
Shift
Meta
Ctrl
Print
Scroll Lock
Pause
Insert
Home
Pg Up
Delete
End
Pg Dn
Num Lock
Home 7
8
Pg Up 9
4
5
6
End 1
2
Pg Dn 3
Insert 0
Delete
Enter



```
move (10) steps
turn ↻ (15) degrees
turn ↺ (15) degrees
point in direction (90 ▾)
point towards [mouse-pointer ▾]
go to x: (0) y: (0)
go to [mouse-pointer ▾]
glide (1) secs to x: (0) y: (0)
change x by (10)
set x to (0)
change y by (10)
set y to (0)
if on edge, bounce
set rotation style [left-right ▾]
```

```
say [Hello!] for (2) secs
say [Hello!]
think [Hmm...] for (2) secs
think [Hmm...]
show
hide
switch costume to [costume2 ▾]
next costume
switch backdrop to [backdrop1 ▾]
change [color ▾] effect by (25)
set [color ▾] effect to (0)
clear graphic effects
change size by (10)
set size to (100) %
go to front
go back (1) layers
```

```
when ⚑ clicked
```

```
when I receive [message1 ▾]
when [space ▾] key pressed
when this sprite clicked
when backdrop switches to [backdrop1 ▾]
when [loudness ▾] > (10)
broadcast [message1 ▾]
broadcast [message1 ▾] and wait
```

```
wait (1) secs
repeat (10)
forever
if < > then
if < > then
else
wait until < >
repeat until < >
stop [all ▾]
```

```
РАЗДЕЛ ПРОЦЕДУР.
НАЧРАБ.
    ОТКРЫТЬ ВХОДНОЙ и т. д.
    ВЫПОЛНИТЬ ЧИСТКА.
ЧТЕНИЕ.
    ПОМЕСТИТЬ НУЛИ В ТАБПЕРЕКЛ.
    ЧИТАТЬ ВХМАСС В КОНЦЕ ПЕРЕЙТИ К ФИНИШ.
    ЕСЛИ ВХГОД НЕ РАВНО РГОД ПЕРЕЙТИ К НОВЫЙ-ГОД.
    СЛОЖИТЬ СБЫТ С РГОДИТОГ.
    ЕСЛИ ВХМЕС НЕ РАВНО РМЕС ПЕРЕЙТИ К НОВЫЙ-
    МЕСЯЦ.
    СЛОЖИТЬ СБЫТ С РМЕСИТОГ.
    ЕСЛИ ВХДЕНЬ НЕ РАВНО РДЕНЬ ПЕРЕЙТИ К НОВЫЙ-
    ДЕНЬ.
    СЛОЖИТЬ СБЫТ С РДНИТОГ.
    ПЕРЕЙТИ К ЧТЕНИЕ.
НОВЫЙ-ГОД.
    ПОМЕСТИТЬ РГОДИТОГ В ВЫХГОДИТОГ.
    ПОМЕСТИТЬ СБЫТ В РГОДИТОГ.
    СЛОЖИТЬ 2 С ТАБПЕРЕКЛ.
НОВЫЙ-МЕСЯЦ.
    ПОМЕСТИТЬ РМЕСИТОГ В ВЫХМЕСИТОГ.
    ПОМЕСТИТЬ СБЫТ В РМЕСИТОГ.
    СЛОЖИТЬ 1 С ТАБПЕРЕКЛ.
НОВЫЙ-ДЕНЬ.
    ПОМЕСТИТЬ РДНИТОГ В ВЫХДНИТОГ.
    ПОМЕСТИТЬ СБЫТ В РДНИТОГ.
```

Рис. 4.5. Табуляция: (*б*) раздел процедур.

```
СМЕНА-ДАТЫ.
    ПОМЕСТИТЬ РДЕНЬ В ВЫХДЕНЬ и т. д.
    ПОМЕСТИТЬ ВХДЕНЬ В РДЕНЬ и т. д.
ПЕЧТАБ.
    ПИСАТЬ ПСТРОКА СПЕРВА 1.
ЧИСТКА.
    ПОМЕСТИТЬ ПРОБЕЛЫ В ПСТРОКА.
ПРОВПЕРЕКЛ.
    ЕСЛИ ТАБПЕРЕКЛ РАВНО 3 ПЕРЕЙТИ К СМЕНА-
    КАЛЕНДАРЯ.
    ЕСЛИ ТАБПЕРЕКЛ РАВНО 1 ВЫПОЛНИТЬ ПЕЧТАБ.
    ПЕРЕЙТИ К ЧТЕНИЕ.
СМЕНА-КАЛЕНДАРЯ.
    ПОМЕСТИТЬ 'ГОД 19' В ВЕК.
    ПОМЕСТИТЬ ВХГОД В ГОД-СТРАНИЦА.
    ПИСАТЬ ПСТРОКА СПЕРВА НОВАЯ-СТРАНИЦА.
    ВЫПОЛНИТЬ ЧИСТКА.
    ВЫПОЛНИТЬ ПЕЧТАБ.
    ПЕРЕЙТИ К ЧТЕНИЕ.
ФИНИШ.
    ПОМЕСТИТЬ РГОДИТОГ В ВЫХГОДИТОГ.
    ПОМЕСТИТЬ РМЕСИТОГ В ВЫХМЕСИТОГ.
    ПОМЕСТИТЬ РДНИТОГ В ВЫХДНИТОГ.
    ПОМЕСТИТЬ РДЕНЬ В ВЫХДЕНЬ и т. д.[1]
    ВЫПОЛНИТЬ ПЕЧТАБ.
    ЗАКРЫТЬ ВХМАСС, ВЫХМАСС.
    ОСТАНОВИТЬ РАБОТУ.
```

[1] В оригинале этот оператор пропущен.—*Прим. перев.*

Рис. 4.5*б*. Окончание.

Access Modifier

Annotate components with information about how others are allowed or not allowed to access them. Access can be limited by inheritance (*protected* in C++), modular structure (*internal* in C#), etc. The most popular modifiers are *public* (everyone welcome) and *private* (fully restricted). Similar modifiers can be used to manage scope, such as *global* and *nonlocal* in Python.

Alphabet §

The basic alphabet is often taken for granted, especially for textual languages, but it is an important design aspect. In some languages (APL being the extreme) the alphabet is extremely broad, with specific symbols being used for built-in operators, which shifts the visual feel of the language closer to mathematics. In other languages keywords are taken from English, which limits language appeal to some groups of users (and may lead to reimplementations with translated keywords).

Assignment

Moving a data from one place to another. Some 4GLs have separate statements for straightforward (byte-copying) and composite (pattern-matching) assignments such as Cobol's *MOVE CORRESPONDING* which requires unification. In modern languages the source data structure (and sometimes the target one) can often be created on the fly. Many languages combine assignment with trivial manipulation (such as +=).

Backtracking

A computation strategy commonly found in declarative languages. Every choice in the evaluation path becomes a *save point* to which the computation returns in case of failure. All the changes made between the save point and the point of failure are undone. Backtracking is common in parsers and logic programming, and used for error recovery everywhere else.

Backward Compatibility

In language evolution, introduce new features that should supercede older ones, but ensure the users that their existing code will still run. Ideally, this code should eventually be rewritten and coevolved.

Block §

Viewing a list of statements as a specific (*compound*) kind of statement is a conceptual eye-opener and allows to treat composite constructs in a uniform and orthogonal way (*if … begin … end* and *do … begin … end* instead of *if … endif* and *do … enddo*). Languages either use delimiters (begin/end or curly brackets) or indentation. Blocks can be seen as degenerate subprograms and be useful in optimisation.

Branching

Forking the computation based on conditions known at runtime, is a popular construct. Control flow can be transferred unconditionally (*branch, jump, goto*), or conditionally (based on true/false, zero/positive/negative, explicit condition, exhaustive patterns, etc). In some languages branching can be done by *guarding* statements with constraints.

Character Type

A family of value types that can be used in a language: single characters, special characters, zero-terminated strings, fixed length strings, variable length strings, structured strings, etc.

Class

A class or a trait represents a template that can be followed by objects: a particular collection of properties and methods that can be always relied on. A class can be then *instantiated* with appropriate parameters to form an object that *conforms to* the class definition. Classes are the ultimate form of encapsulation. They can be inherited from one another to form subclasses.

Client/Server

A language may allow one conceptual model to be split into two intercommunicating components to be executed in parallel: the *server* side which has access to all the necessary system data and runs in a fully controlled environment, and the *client* side which runs closer to the system user's data and has to survive in a much less controllable environment. Client code and server code can be written in different languages or compiled to different languages before deployment.

Code Completion

Many IDEs monitor what the language user is typing and make suggestions based on their knowledge of the language keywords, constructs allowed in the context, variables visible from the current namespace, etc. The list of such suggestions must be short to be useful, otherwise it does nothing but annoy the users.

Code Generation

Generation of machine code, intermediate code, a model in a target language, an output model or a textual result, is the last phase of a classic compiler (before or after optimisation). What is typical for code generation is the richness of the input (generously annotated intermediate graphs) and a deliberate limitedness of the output (which is often platform-specific and/or hardware-specific). In MDE code generation is usually implemented by model-to-text transformations.

Code Mining

Besides user surveys and expert opinions, there is a third way to uncover points to improve the language in its next versions: examining existing models created in this language. There are many modern techniques in mining software repositories that can be helpful here: clustering, vocabulary analysis, statistics (especially correlations), natural language processing, information retrieval, machine learning, etc.

Code Ownership

Signing the user's name under a piece of code has the same effect as signing a person's name on an item: caring about what happens to the item later. Comments explaining which dev made which code changes existed since very early on. In modern ecosystems, ownership is tracked automatically by a version control system and can be checked at any time (*git blame*).

Collection

Arrays, lists, tuples, sets and multisets are the most common composite user-defined parametrised types for *collections* of elements. It is up to the language designer to decide which ones are supported and how they are handled — can elements within one collection have different types, are they mutable, passed by name/value /reference, etc.

Comment

Comments are pieces of documentation built directly into the source of the system. Most IDEs support comments visually by presenting them in a completely different colour, usually dimmer than the rest of the model, to focus developers on executable constructs first. In some languages like BibTeX or INTERCAL everything uncompilable is a comment. Some comments like codetags, Javadoc or Documentation Comments are strictly or semi-structured.

Compilation Error

Modern languages have many means of assessing validity of the model before it is actually used. Thus, compilers tend to have a sophisticated error handling facility and try to provide enough information for the language user to fix the problems. Some languages are notoriously known for providing bad error messages. There are many ways to *recover* from an error in order to analyse the rest of the program and report multiple problems at once. Can be provided as a live feedback.

Compilation Warning

When a compiler detects a possibly dangerous situation with extremely limited applicability, it displays a warning message and proceeds with the build process anyway. In many cases there is a special option for disabling a particular warning for a particular piece of code. Warnings can be given when an *anomaly* or a smell is detected, and may involve some form of error correction. Can be provided as a live feedback.

Comprehension §

List and set comprehensions are language constructs resembling the mathematical notation for creating a set by its characteristic function (*"for all numbers from 1 to 10, give me their squared values"*), and combine *map* and *filter* classical for functional programming. Comprehensions as a language construct exist in Haskell, Python, Rascal, C# and some other languages.

Concrete Syntax

The way to describe the concrete representation of the programs. The concrete syntax is used by humans to read, write, create and understand sentences of the language. Usually the only languages that do not have concrete syntax are those intended for internal intermediate representation. Some languages have more than one.

Concurrency

Since modern computers and systems are good at multitasking, a language designer may decide to use that. An executable model can then be decomposed into components that are executable in parallel on different CPU cores or different devices. This can be completely undesirable (to avoid deadlocks, overhead, race conditions, etc), or performed automatically, or use the language user's guidance in synchronisation of threads, tasks and processes.

Constraint

Besides languages which programs are expressed only in terms of constraints (OCL, CLP(R), Oz), there are many that have them in one form or another. The most popular form is *assertions*, a non-invasive form of exception handling allows language uses to explicitly state (*assert*) invariants, pre-conditions and post-conditions as logic expressions that must universally hold. Such assertions can be easily removed before deploying the system into production.

Cross-compilation

A cross-compiler works on one platform but ultimately targets another. Relying on a cross-compiler allows to separate the development platform from the one where the programs get deployed to — for instance, a mobile app developer can work with a proper keyboard and a big screen. The IDE for a cross-compiled language may include a virtual machine for execution, debugging, etc. A compiler capable of producing code for different targets, is called *retargetable*.

Debugging

The activity of finding and fixing sources of incorrect behaviour is not enjoyed by many language users, but is used by all of them without exception anyway. Declarative and constraint languages are the hardest to debug due to their complex evaluation strategies (unification, backtracking, etc) and imperative ones are the easiest since they specify the algorithm most explicitly. Most modern languages are shipped with a dedicated debugger or have debugging functionality in the IDE.

Default

Unchanged configuration options, uninitialised variables and unspecified optional modifiers are examples of situations when a default value must be used by the compiler. These default values are decided by the language designer and typically represent the *best* option within the paradigm.

Deployment

Once the model written in a language, has been checked, compiled, linked and otherwise prepared for use, it may need to be *deployed*. This happens directly by copying it to the machine of the end user, or by connecting it to the network, or by creating a special installer, etc. In many cases deployment is not viewed as a concern of a language designer, but among practitioners it is perceived as a part of language design.

Deprecated Construct

In language evolution, sometimes a no longer desired construct cannot be simply removed to avoid breaking backward compatibility. However, it can be marked explicitly as deprecated to discourage language users to rely on it.

Design Chart/Diagram

UML distinguishes between structural (class, package, object, component, composite structure, deployment) and behavioural diagrams (activity, sequence, use case, state, communication, interaction overview, timing). The former specify and visualise structure breakdown, the latter — events and interaction. Some languages (e.g., syntactic diagrams) are both.

Documentation

There are two equally important kinds of language manuals: for people learning the language and for its active users — and sometimes these are two disjoint sets of documents. Documentation may contain executable examples and can/should be automatically checked for internal validity and consistency. Some documentation elements must be provided through an IDE, especially if the language is an API.

Encapsulation

Most high level language abstract from low level details like video memory access, memory allocation, register values, caching, etc. Depending on the language design and philosophy, these features may be prohibited or just hard to find for beginners. Data structures can also be encapsulated by bundling them into records or classes, and code can be organised in hierarchical modules and subprograms.

Energy Saving

Computationally heavy code requires more CPU or GPU cycles, which consumes more power, which in turn makes the applications spend more energy. Making a compiler of a language especially optimised towards power reduction may increase its appeal for users that intend to run their programs on devices with limited power (mobile phones and smaller). Power reduction and energy saving techniques also contribute towards global sustainability, and can be used/chosen for ethical reasons.

Enumeration Type

An enumeration is a data type that defines a very limited set of possible values which are, nevertheless, more comfortably referred to by their names and not by encoded numbers. The most famous enumeration is the Boolean (logical) type, which contains only two values: *true* and *false*. If the domain permits, the language does not have to support user-defined enumerations.

Esotericism

INTERCAL, Unlambda, Befunge, Malbolge and other esoteric languages are based on paradigms so unconventional that writing even one program puts disproportional strain on the users. This challenging nature makes people engage and compete in programming in such languages as a form of entertainment. LOLCODE, ArnoldC and others are languages developed based on the memes that are circulating among software engineers: the popularity of them piggybacks entirely on the viral nature of those memes.

Event

The first implementations of user interfaces were turning the entire program into a giant loop waiting for the user to activate its functionality by choosing the way to communicate (click, tap, edit, etc). Since direct implementations of such an event loop are not green (consume too much energy), event handling can be built natively into the language and implemented efficiently by the compiler and hardware. Events are used for interacting with end users, sensors, threads, etc.

Exception Handling

An emergency sibling of branching used for extraordinary situations — can be slower than the normal branching, but usually more robust in handling situations like a cricial failure during the handling of another failure. A less invasive form of exception handling are assertions.

Execution Error

Errors can happen at compile time, but also at run time, due to hardware faults, communication problems, invalid user input or simply bugs that were left undetected at compile time by static analysis. Some languages (Erlang) have very well-designed strategies for handling execution errors, but all others also feature some form of partial recovery from them. The language user controls runtime error handling with exceptions.

Expressivity

There is ultimate expressivity of a software language, typically incorporated in answers to questions like *"is it Turing complete?"* (i.e., does it have enough constructs to emulate a Turing machine?), and there is a much more important and subtle issue of local expressivity in the sense of how small programs can get without sacrificing their readability. Many languages eventually develop shorthand constructs for writing commonly used combinations of constructs shorter and thus faster.

First Class Citizen

It is an important design point to decide which entities within a program have the right to be saved, passed as arguments, transferred through other means, etc. Numbers? Collections? Objects? Functions? Unfinished computations? Data streams? Unfilled templates?

Garbage Collection

Automatic release of memory is impossible for cyclic data structures. Languages that want to support them, have a *garbage collector* — a runtime compiler component that occasionally *marks* data structures that have become inaccessible and then *sweeps* them away, freeing the memory. GC can compromise language responsiveness and performance.

Generation

Tedious, repetitive and error-prone programming tasks can be automated by using templates, wizards, explicit staging/morphing constructs of generative programming, construction workbenches, etc. In many practical cases the language user is allowed to edit the result to fine-tune it. The final generation phase is called code generation.

Heterogeneous Data

Some languages allow considerable freedom in types that makes collections capable of carrying elements of varying structure. Examples: variant records in Modula and Ada, heterogeneous lists in Python, polytypic functions in Haskell, GADTs in Haskell. Allowing heterogeneity empowers the language user but makes the language harder to learn.

IDE

Integrated Development Environments (IDEs) are used to support language users in their common tasks: code navigation, debugging, building, modularising, refactoring, etc. Can take a form of a dedicated standalone editor, a website or a plugin for a universal editor. Needs to have a well-designed UI.

IDE GUI

Most IDEs divide the screen space among areas with different functionality: for navigating through adjacent models, for editing the code, for reviewing the architecture, for watching how values change at runtime, etc. Advanced IDEs like IntelliJ, Eclipse or VS.NET have so many subwindows that the user has to choose which ones to keep open at each given time.

Indentation & Whitespace

The two extremes for this aspect are: treat indentation as something crucial to the program structure (and thus process constructs differently based on columns where they start) and discard all possible indentation (even in the middle of names, as FORTRAN does). Most languages are somewhere in the middle. Normalisation of whitespace use is called pretty-printing.

Inheritance

An "is-a" relation can be represented by a language construct when one class, object or function *inherits* all the properties of its *parent* and possibly adds others exclusive to itself. It is a design consideration which entities can be derived from which, what are the rules for inheriting from several parents, etc.

Input/Output

Most executable models are not self-contained and require input data to run and produce results, which in turn need to be propagated somewhere. There are languages that are volatile with input and output, those that only work with files, those that wrap I/O as a side effect of a monad, etc.

Instruction Set

Instead of freely combinable statements and expressions, low level languages (microcodes, assemblers, virtual machine bytecodes, etc) have limited non-extendable instruction sets. Each of the instructions typically has a mnemonic (name) and a bit-level encoding. Realistic assemblers had to introduce macro expansions to make expressivity and programming experience tolerable.

Iteration

There are many looping constructs, ranging from the imperative classics such as a for loop, to the functional classics such as *map*, *filter* and *fold* (or *reduce*). It is not uncommon for languages to support only some of these constructs. Some older GPLs and 4GLs also have one iterative construct which can be annotated with all kinds of conditions and steppers.

Keyword

Special words in concrete syntax of the language that carry identical meaning across all possible models in the same language. Can be made reserved so that programmers may not redefine them. A language can get new keywords by evolution.

Labelling

Since most engineers know several languages, some language manuals directly assume initial familiarity of their users with other languages. Can refer to paradigms or families (*"this language is strongly typed"*) or directly to other languages (*"inheritance works like in Java"*). Also, by explicitly stating which camp the language is siding with or which key community figures endorse it, the designer can invoke an emotional response directly mappable to language's acceptance and popularity.

Lazy Evaluation

A *lazy* compiler defers evaluation to the latest possible moment. Lazy languages allow infinite data structures (as long as they are processed one chunk at a time) and may have unpredictable outcomes if calculations are allowed to have side effects (like C's ++). Lazy evaluation has many applications from optimisation of generated code to stream data processing.

Live Feedback

An advanced IDE running on modern hardware can utilise its idle cycles to attempt parsing, compilation, dependency analysis and other kinds of checks while the language user is still typing the model. Erroneous and suspicious pieces of code are commonly underlined with red or yellow squiggly lines familiar from natural word processors.

Lock-out/Opt-in

Certain combinations of language features may be disabled (erroneous) by default, with a possibility of enabling them explicitly. For example, redefining a method in a derived class is only allowed in C# when a specific *override* keyword is used, which leaves visual cues to future readers of the piece of code in question.

Macro

A mechanism commonly found in low level languages that allow users to define a piece of syntactic sugar to be *expanded* into a longer sequence of instructions. Advanced parametrised macros resemble subprograms in expressivity but may behave less reliably due to their lexical nature. In bigger languages macros are typically handled by a preprocessor.

Metaphor

Give your language constructs names that need no explanation: atom, backtracking, binding, body, build, cloud, collision, compiler, dangling else, debugging, desugaring, dictionary, duck typing, environment, filter, floating point, forest, framework, garbage collection, go to, heap, inheritance, jump, library, linking, map, pointer, pruning, rendezvous, stack, turtle, weaving, window, …

Module

Large models inevitably outgrow their creators' capabilities to understand them all at once. Comprehension can be aided greatly by the language providing modules, packages, classes, procedures, blocks and other elements to group related code fragments together. Modern IDEs can analyse code for cohesion and coupling to help improve modularisation. Modules are often [one of the possible] *compilation units*.

Natural Pattern

Design patterns, implementation patterns and architecture patterns are used across language boundaries, but many domain-specific languages incorporate well-known patterns as native language constructs: Model-View-Controller, Singleton, State, Visitor, etc.

Numeric Data Type

Often gets overlooked at the early stages of language design, but could significantly shape the application area of the language. There are many integer types, distinguished by their byte sizes and therefore value ranges; also decimal types with fixed scale and precision; and floating point types good for scientific computations but not for handling finances.

Operator Overloading

A language designer may decide to reuse the same symbol for several different operators, usually conceptually related (such as + for arithmetic addition and string concatenation). Using it for totally unrelated operations is considered harmful for readability (such as & for pointer referencing and bit conjunction in C). In some languages (C++, Ada, Fortran) language users can also redefine their own operators that complement their own defined types.

Operator Precedence

To avoid excessive use of parentheses, a language can provide a default convention of disambiguating constructs with 3+ entities bound by binary operators. In arithmetic expressions, the precedence usually follows mathematical laws.

Optimisation

It is always easier and less error-prone to generate intermediate code or machine code with simple and straightforward patterns and subsequently optimise the result in a different phase. The effect on the language users is that they do not need to optimise their models to the fullest, since their own naive code will be optimised together with the rest. Small efficiences are only relevant 3% of the time, for the rest premature optimisation is considered the root of all evil.

Order

Many languages have ordering constraints: a variable must be declared before its use, a function signature known before its call, etc. Sometimes constructs are grouped and it is the groups that must follow the order: e.g., first all declarations, then all functions, then the rest of the code (COBOL's *divisions* are the extreme example of this).

Orthogonal Design

Independent features should be controlled by independent mechanisms. Related constructs should look similar and different ones should look different. Regular rules without exceptions are easier to learn. The fewer surprises one has while learning the language, the higher the language quality.

Parameter Passing

There are several strategies in mapping *arguments* that are being passed to a procedure in a call with the *parameters* that procedure expects to get: *call by value* (expose only the values, safe but inefficient for composite data), *call by result* (can return several values at once), *call by value-result* (the caller gets values, updates them, they are passed back), *call by reference* (expose pointers to values, efficient but unsafe), *call by name* (evaluate pointers when they are used inside the caller), etc.

Parametrised Type

Some types can be defined partially by the user and partially by the language designer. For example, the language *designer* knows what a list is, and the language *user* can select any other type for list elements — this will change handling of such elements, but the philosophy behind their collection will stay the same.

Performance

Performance testing and its variations like profiling and stress testing are commonly desired nice-to-have features in IDEs. Languages and their ecosystems greatly vary in the extent to which this aspect is recognised and supported.

Phased Process

Breaking a process into phases is one of the most used divide-and-conquer principles applied in language processing. Most compilers are designed to work in phases, and different competences and skills are required to implement each phase.

Picture Clause

A data type that saves a specially formatted entity (usually a float or a date) that can be used directly in printing statements but also manipulated as data.

Platform Lock-in/out

Supporting a great language only for one particular hardware platform, OS or IDE, implicitly forces people to use them. For example, malware practices of Java installers turned some users agains JVM, which also deprived them of Scala and Clojure. Another example is .NET Core, a redesign of the .NET Framework which allows typically Windows-specific code to run on Linux.

Pointer

A popular data type in low level languages, representing a memory address where the data structure is stored — which is more efficient to pass across functions than the structure itself. The type of the structure needs to be known to decipher itse contents, since the pointer itself is nothing more than a number.

Pretty-printing

A language can have a default formatting convention that is not only accepted by the community to improve the representation quality of the models, but also automated and shipped in a form of a tool. Such a tool can be very configurable, have limited feature selection or none at all. A pretty-printer that scans the input and minimises the delimiters in it, is sometimes called a program *compactor*. Pretty-printers are omnipresent in textual languages and may require layout strategies/policies for graphical ones.

Preview

Some features are very useful in general, but implemented in a way that sometimes fails. In this case, the impact of an application of a feature can be explicitly examined by the language user before agreeing to proceed. Common for database queries and object-oriented refactorings.

Product Line

Unified production of things that belong to the same family (similar cars, computers, furniture, drinks) is typical for any industry, including software engineering. To make it so that one codebase specifies the behaviour of a system that must be compiled and deployed under a variety of devices and hardware architectures, the model can be annotated with conditions to be checked during compile time and result in different code to be produced by the compiler to be run later. To simplify compiler construction, conditional compilation is handled by a preprocessor.

Readability

In many cases a model is written once but read many times for many purposes: to fix bugs, introduce new features, understand its behaviour, change it, etc. Hence, the ease of reading a model can become a cornerstone of the design of a language. Some existing languages, like COBOL and many modern DSLs, were specifically designed to empower domain experts to read and write in them. Others, like APL and Perl, while expressive, are known to produce unreadable models.

Record

Many languages have some kinds of records or structures that bundle several related pieces of data without attaching methods to work with that data. A dynamic variation thereof is known as a *dictionary* or a *map* (e.g., hashmap) and it allows users to add and remove fields at runtime.

Redefine

Once something has been defined, it can be redefined in many languages: derived classes can overload base classes' properties, local variables can shadow the global ones with the same name, the same memory fragment can be treated as belonging to two separate data types (requiring *alias analysis*), etc.

Refactoring

Refactorings are code changes that do not impact the system's behaviour but change its internal structure to improve code quality, prepare for the subsequent change, etc. Some DSMLs mean their models to *only* change through refactorings and refinements. In other domains it is also not uncommon to eventually get refactoring support in the IDE (often with previews).

Runtime

A runtime environment is a system component that must accompany the result of the compilation in order for it to function correctly. May be completely non-existent, contain standard libraries or a virtual machine. The *multi-stage* paradigm can accommodate several intermediate stages of generation between compile-time and runtime.

Scope & Binding

If a type or a variable is declared, how far from the declaration can you still use them? If an outside entity is used in a subprogram, will it be taken from the parent scope of the subprogram or from the scope of the call? Several equally viable paradigms are known for scoping, name-type binding and declaration-reference binding.

Security

Security concerns were added to most mainstream programming and modelling languages only after their initial release, and fit there with varying degree of comfort. When designing a new language, it is possible to build in constructs for policy management, encryption, authentication and others, right from the start.

Smell

A *smell* is found when a code fragment has suspicious characteristics even without being wrong. Modern computer science identifies smells for code, models, architecture, spreadsheets, hardware, grammars, etc. The very use of the word "smell" strongly implies that any neat self-respecting language user should try to leave less of those when touching the code.

Standard Library

A library or a set of libraries that are built into the language. It can be a tough design decision for the language designer to decide which functionality needs to become native constructs and which can go into the standard library. Such library functions can be expanded into code directly or shipped together with the compiled code as a runtime.

Static Analysis

If a language does not directly limit its users' ability to express thing in a "bad" way, this is still possible for the compiler of the language. Parsing, type analysis, dependence analysis, formatting, conventions are all examples of this. One of the language design principles states that if an error gets though one line of defence, it should be caught by the next one.

Sublanguage

A language can consist of several smaller languages that model its separate aspects, or incorporate other languages for solving specific subtasks. For example, a programming language may embed a database query sublanguage, or a specification language may embed a constraint language. An embedded language can be reinvented to fit its context or reused by adopting an existing one that has already demonstrated its usefulness.

Subprogram

Pieces of code executable from other places promote reuse, but can be designed differently. Some languages only allow them to be independent (*procedures, functions*), or attached to an object (*methods*) or a class (*static methods*), others provide special synchronisation among them (*coroutines, delegates*) or allow them to be passed as arguments. If they may have *parameters*, the designer must decide on the parameter passing strategy.

Substitution

When a subprogram specifies the types of input it expects, these types do not need to be treated precisely: often one can use entities of subtypes of the specified types (e.g., put a circle in a function that draws a shape because a circle is a subtype of shape). Subtyping is nontrivial, and the designer must choose among covariance, contravariance, invariance, etc. Non-strict handling of values of different types involves designing rules for type casts and conversions.

Synchronisation

Managing the use of resources by some predefined form of synchronisation between readers and writers. Can be synchronous or asynchronous, and take forms of resource locks, semaphors, pipes, rendezvous, handshakes, message passing channels, etc. Always needed for concurrent computing.

Syntactic Sugar

Nice-to-have constructs that are not extending the expressive power of the language, are sometimes not actually implemented directly — just expanded into bigger sequences of more primitive and less user-friendly constructs.

Syntax Highlighting

A development environment of the language can profit from visualisation even if the language is textual by colour-coding different categories of words (strings, numbers, standard libraries, reserved words, etc). Colours are also commonly used for non-textual languages, unless models are expected to be printed or viewed on grayscale devices.

Trade Off

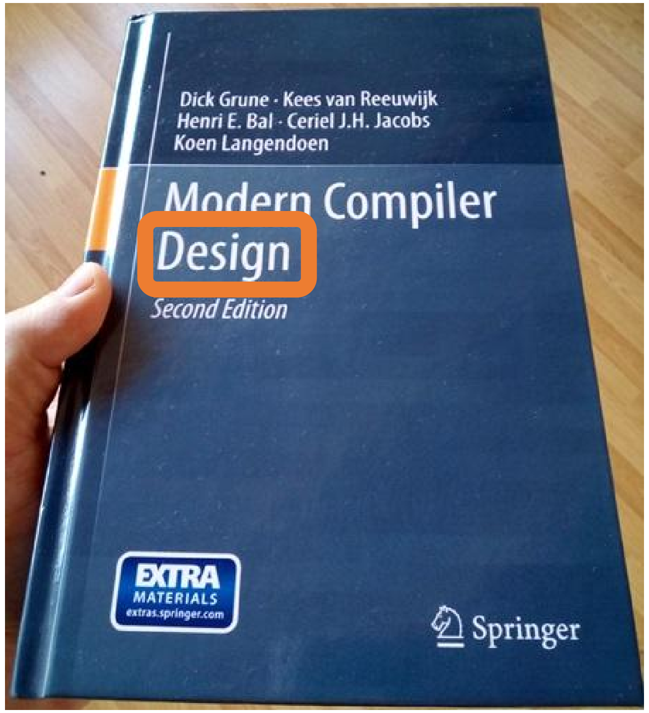
When direct optimisation is impossible or not sufficiently effective, the language designer can identify trade offs and leave them all inside the language for the users to choose. For example, many compilers have compilation options optimising code generation for speed, size or power, but not all three. Many languages have libraries and structures for both *arrays* (fast, immutable length) and *lists* (slower but flexible).

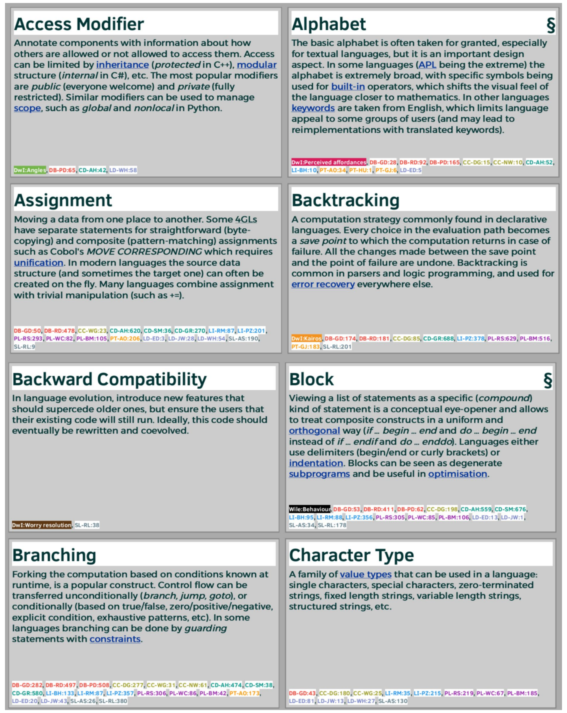
DYOL: Design Your Own Language

Access Modifier
Alphabet §
Assignment
Backtracking
Backward Compatibility
Block §
Branching
Character Type
Class
Client/Server
Code Completion
Code Generation
Code Mining
Code Ownership
Collection
Comment
Compilation Error
Compilation W
mplementation §
Concrete Syntax
Concurrency
Constraint
Cross-compilation
Debugging
Default
Deployment
Deprecated Construct
Design Chart/Diagram
Documentation
Encapsulation
Energy Saving
Enumeration Type
Esotericism
Event
Exception Handling
Execution Error
Expressivity
First Class Citizen
Garbage Collection
Generation
Heterogeneous Data
IDE
IDE GUI
Indentation & Whitespace
Inheritance
Input/Output
Instruction Set
ation
Keyword
Labelling
Lazy Evaluation
Live Feedback
Lock-out/Opt-in
Macro
Metaphor
Module
Natural Pattern
Numeric Data Type
Operator Overloading
Operator Precedence
Optimisation
Order
Orthogonal Design
Parameter Passing
Parametrised Type
Performance
Pretty-printing
Preview
Product Line
Readability
Record
Redefine
Smell
Sta rd brary
tic Analysis
Sub gram
Substitution

http://slebok.github.io/dyol

Conclusion





Access Modifier

Annotate components with information about how others are allowed or not allowed to access them. Access can be limited by inheritance (*protected* in C++), modular structure (*internal* in C#), etc. The most popular modifiers are *public* (everyone welcome) and *private* (fully restricted). Similar modifiers can be used to manage scope, such as *global* and *nonlocal* in Python.

DwI:Angles, DB-PD:65, CD-AH:42, LD-WH:58

Alphabet §

The basic alphabet is often taken for granted, especially for textual languages, but it is an important design aspect. In some languages (APL being the extreme) the alphabet is extremely broad, with specific symbols being used for built-in operators, which shifts the visual feel of the language closer to mathematics. In other languages keywords are taken from English, which limits language appeal to some groups of users (and may lead to reimplementations with translated keywords).

DwI:Perceived affordances, DB-GD:28, DB-RD:92, DB-PD:165, CC-DG:15, CC-NW:10, CD-AH:52, LI-BH:10, PT-AO:34, PT-HU:1, PT-GJ:6, LD-ED:5

Assignment

Moving a data from one place to another. Some 4GLs have separate statements for straightforward (byte-copying) and composite (pattern-matching) assignments such as Cobol's *MOVE CORRESPONDING* which requires unification. In modern languages the source data structure (and sometimes the target one) can often be created on the fly. Many languages combine assignment with trivial manipulation (such as +=).

DB-GD:50, DB-RD:478, CC-WG:23, CD-AH:620, CD-SM:36, CD-GR:270, LI-RM:87, LI-PZ:201, PL-RS:293, PL-WC:82, PL-BM:105, PT-AO:206, LD-ED:3, LD-JW:28, LD-WH:54, SL-AS:190, SL-RL:9

Backtracking

A computation strategy commonly found in declarative languages. Every choice in the evaluation path becomes a *save point* to which the computation returns in case of failure. All the changes made between the save point and the point of failure are undone. Backtracking is common in parsers and logic programming, and used for error recovery everywhere else.

DwI:Kairos, DB-GD:174, DB-RD:181, CC-DG:85, CD-GR:688, LI-PZ:378, PL-RS:629, PL-BM:516, PT-GJ:183, SL-RL:201

Backward Compatibility

In language evolution, introduce new features that should supercede older ones, but ensure the users that their existing code will still run. Ideally, this code should eventually be rewritten and coevolved.

DwI:Worry resolution, SL-RL:38

Block §

Viewing a list of statements as a specific (*compound*) kind of statement is a conceptual eye-opener and allows to treat composite constructs in a uniform and orthogonal way (*if … begin … end* and *do … begin … end* instead of *if … endif* and *do … enddo*). Languages either use delimiters (begin/end or curly brackets) or indentation. Blocks can be seen as degenerate subprograms and be useful in optimisation.

Wile:Behaviour, DB-GD:53, DB-RD:411, DB-PD:62, CC-DG:198, CD-AH:559, CD-SM:676, LI-BH:95, LI-RM:88, LI-PZ:356, PL-RS:305, PL-WC:85, PL-BM:106, LD-ED:13, LD-JW:1, SL-AS:34, SL-RL:178

Branching

Forking the computation based on conditions known at runtime, is a popular construct. Control flow can be transferred unconditionally (*branch, jump, goto*), or conditionally (based on true/false, zero/positive/negative, explicit condition, exhaustive patterns, etc). In some languages branching can be done by *guarding* statements with constraints.

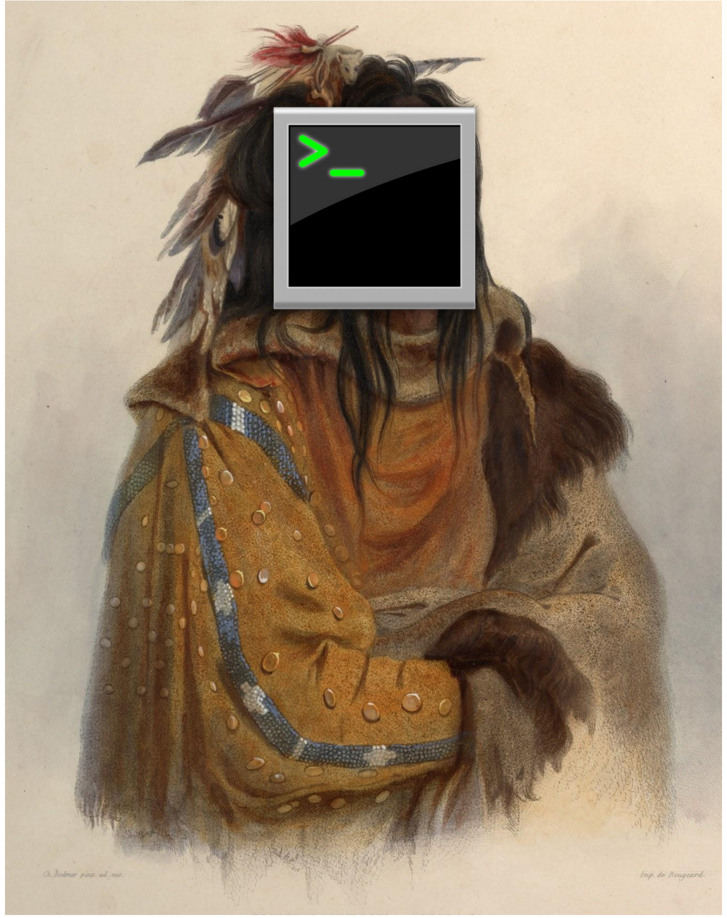
DB-GD:282, DB-RD:497, DB-PD:508, CC-DG:277, CC-WG:31, CC-NW:61, CD-AH:474, CD-SM:38, CD-GR:580, LI-BH:133, LI-RM:87, LI-PZ:357, PL-RS:306, PL-WC:86, PL-BM:42, PT-AO:173, LD-ED:20, LD-JW:43, SL-AS:26, SL-RL:380

Character Type

A family of value types that can be used in a language: single characters, special characters, zero-terminated strings, fixed length strings, variable length strings, structured strings, etc.

DB-GD:43, CC-DG:180, CC-WG:25, LI-RM:35, LI-PZ:215, PL-RS:219, PL-WC:67, PL-BM:185, LD-ED:81, LD-JW:13, LD-WH:27, SL-AS:130

Software Language Engineering Body of Knowledge



DYOL

a toolkit for software language design with intent

[MoDELS’17]

Grammar Zoo

a collection of grammars in a broad sense (mms)

[SCP 2015]

BibSLEIGH

a literature exploration platform

[SATToSE’15]

GraSs

a taxonomy of smells in grammars in a broad sense

[SLE’17]

Picture credit

- http://slebok.github.io/dyol/ (CC-BY, Vadim Zaytsev)
- https://commons.wikimedia.org/wiki/File:Arts_et_Metiers_Pascaline_dsc03869.jpg (CC-BY-SA, David Monniaux)
- https://commons.wikimedia.org/wiki/File:Hand-driven-jacquard-loom.jpg (CC-BY-SA, Edal Anton Lefterov)
- https://commons.wikimedia.org/wiki/File:M0354_1951-40-004_2.jpg (CC-BY-SA, Jacques Monnin)
- https://commons.wikimedia.org/wiki/File:Flow_chart_of_Planning_and_coding_of_problems_for_an_electronic_computing_instrument,_1947.jpg (PD)
- https://commons.wikimedia.org/wiki/File:Swift_logo_with_text.svg (PD)
- https://github.com/dart-lang/logos (CC-BY)
- https://commons.wikimedia.org/wiki/File:Zootfly_brainstorm.jpg (PD)
- https://www.pexels.com/photo/man-planting-plant-169523/ (CC0)
- https://commons.wikimedia.org/wiki/File:Blackfoot_chief_and_Peikann_chief_0078v_cropped.jpg (PD)
- Fair Use:
  - http://www.omg.org/spec/UML/2.5/ (Figure 14.36 from page 334)
  - https://www.oracle.com/us/assets/javaone-logo-guidelines-2211236.pdf
  - https://www.python.org/community/logos/
  - http://beststickers.net/php-elephant-computer-vinil-stickers-decal
  - https://worldvectorlogo.com/logo/scala-4
  - https://wiki.haskell.org/Haskell_logos
  - https://www.w3.org/html/logo/
  - http://rascal-mpl.org/
  - https://www.eclipse.org/atl/
  - https://eclipse.org/epsilon/
  - http://umlforum.com/
  - https://docs.racket-lang.org/images/Logos.html
  - https://golang.org/
  - https://www.packtpub.com/application-development/implementing-domain-specific-languages-xtext-and-xtend-second-edition
  - https://www.amazon.com/Language-Implementation-Patterns-Domain-Specific-Programming/dp/193435645X/
  - https://www.amazon.com/Software-Language-Engineering-Domain-Specific-Metamodels/dp/0321553454/
  - https://www.springer.com/us/book/9783319188201
  - https://www.amazon.com/DSL-Engineering-Designing-Implementing-Domain-Specific/dp/1481218581
  - https://www.amazon.com/Practical-API-Design-Confessions-Framework/dp/1430209739
  - https://www.amazon.com/Build-APIs-You-Wont-Hate/dp/0692232699/
  - http://designwithintent.co.uk/
  - http://www.stroustrup.com/
  - http://www.bjfogg.com/
  - *Black Sails* via https://www.pinterest.com/pin/366621844471744481/