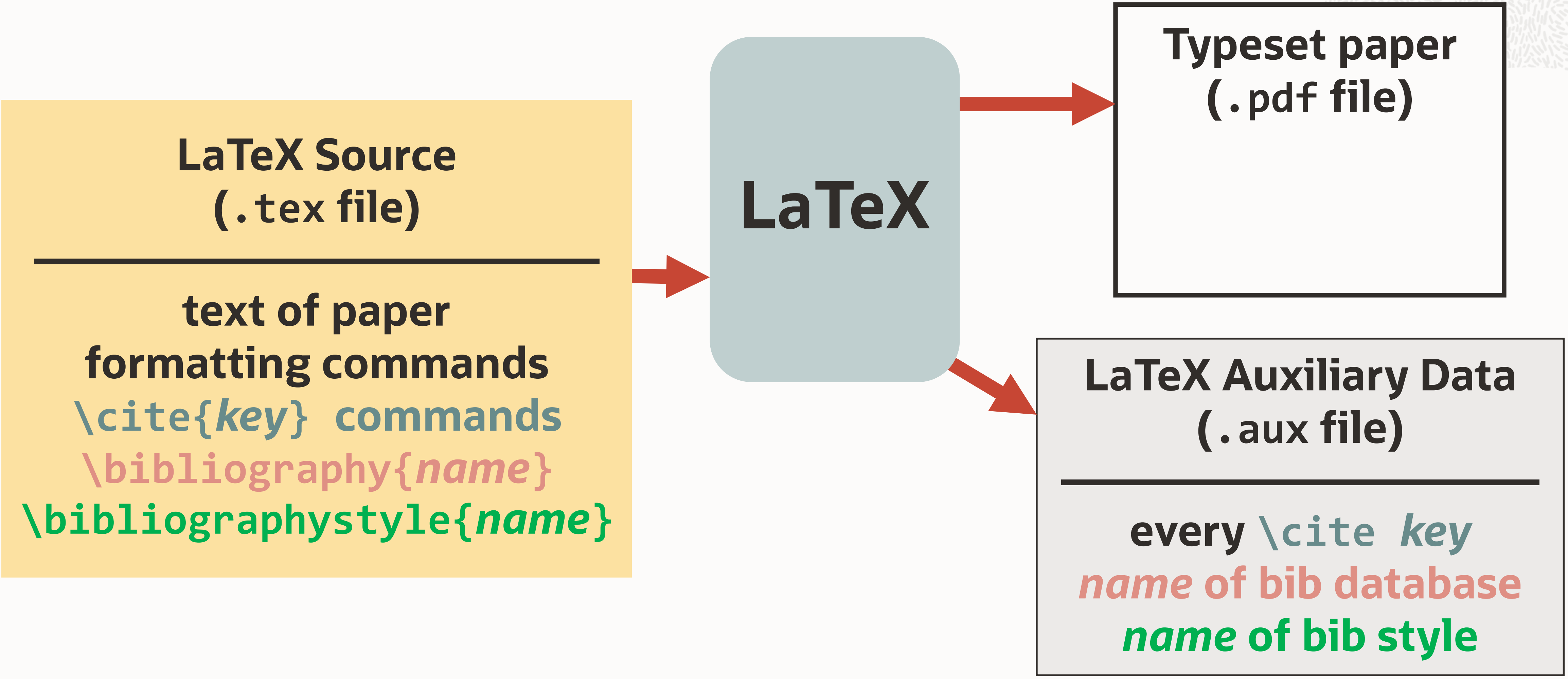
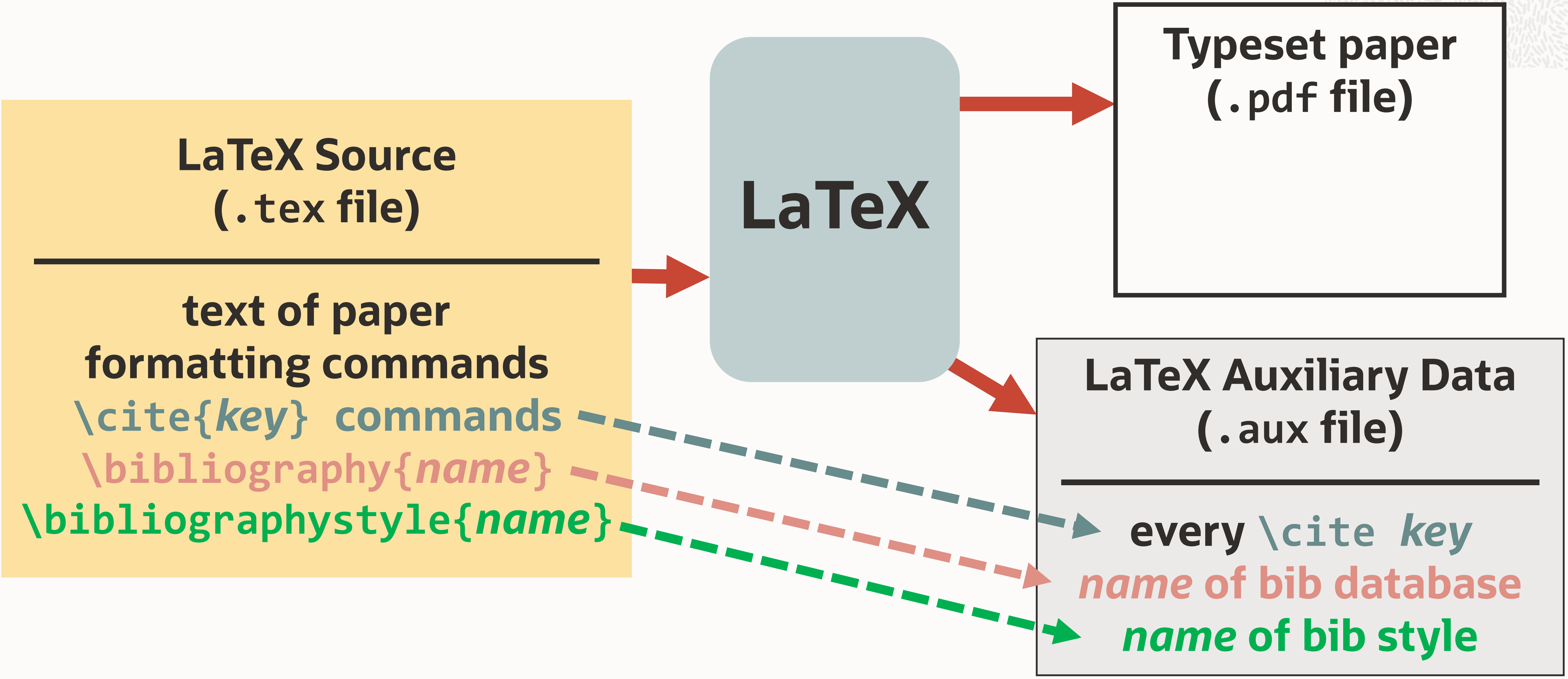
Generality—or Not—
in a Domain-Specific Language
(A Case Study)

Guy L. Steele Jr.
Software Architect
Oracle Labs
March 26, 2021

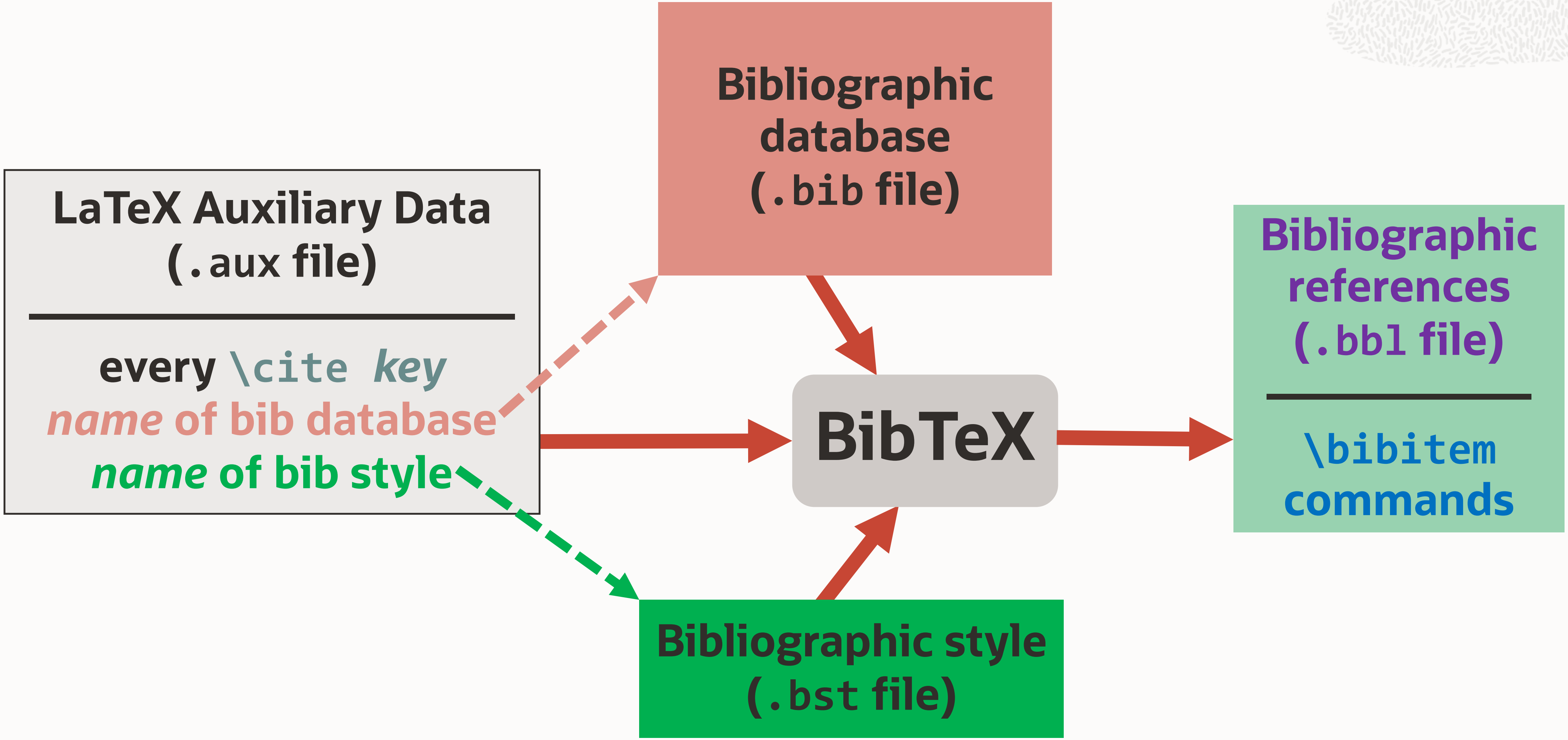
How BibTeX Works



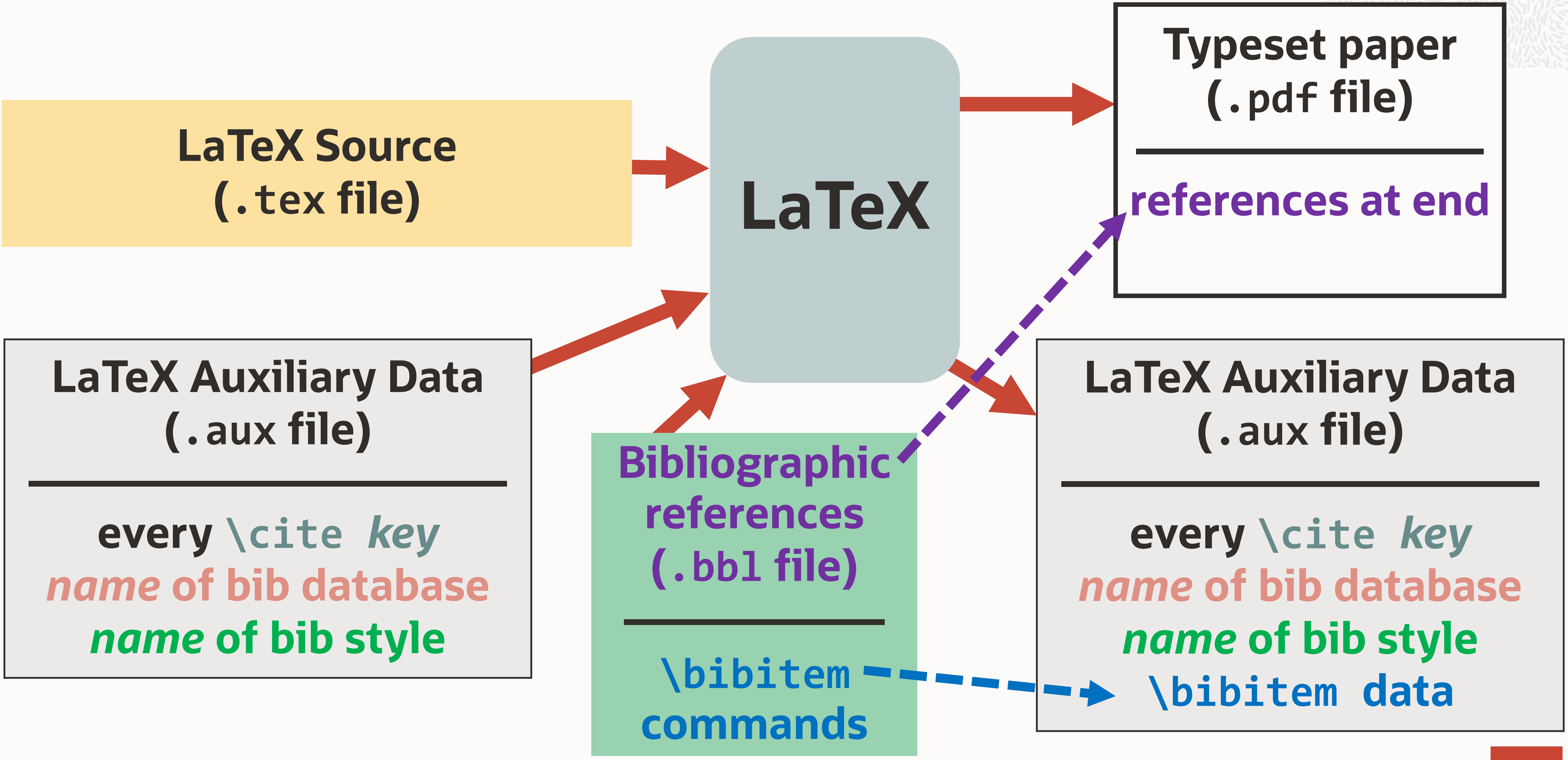
How BibTeX Works: First pass through LaTeX



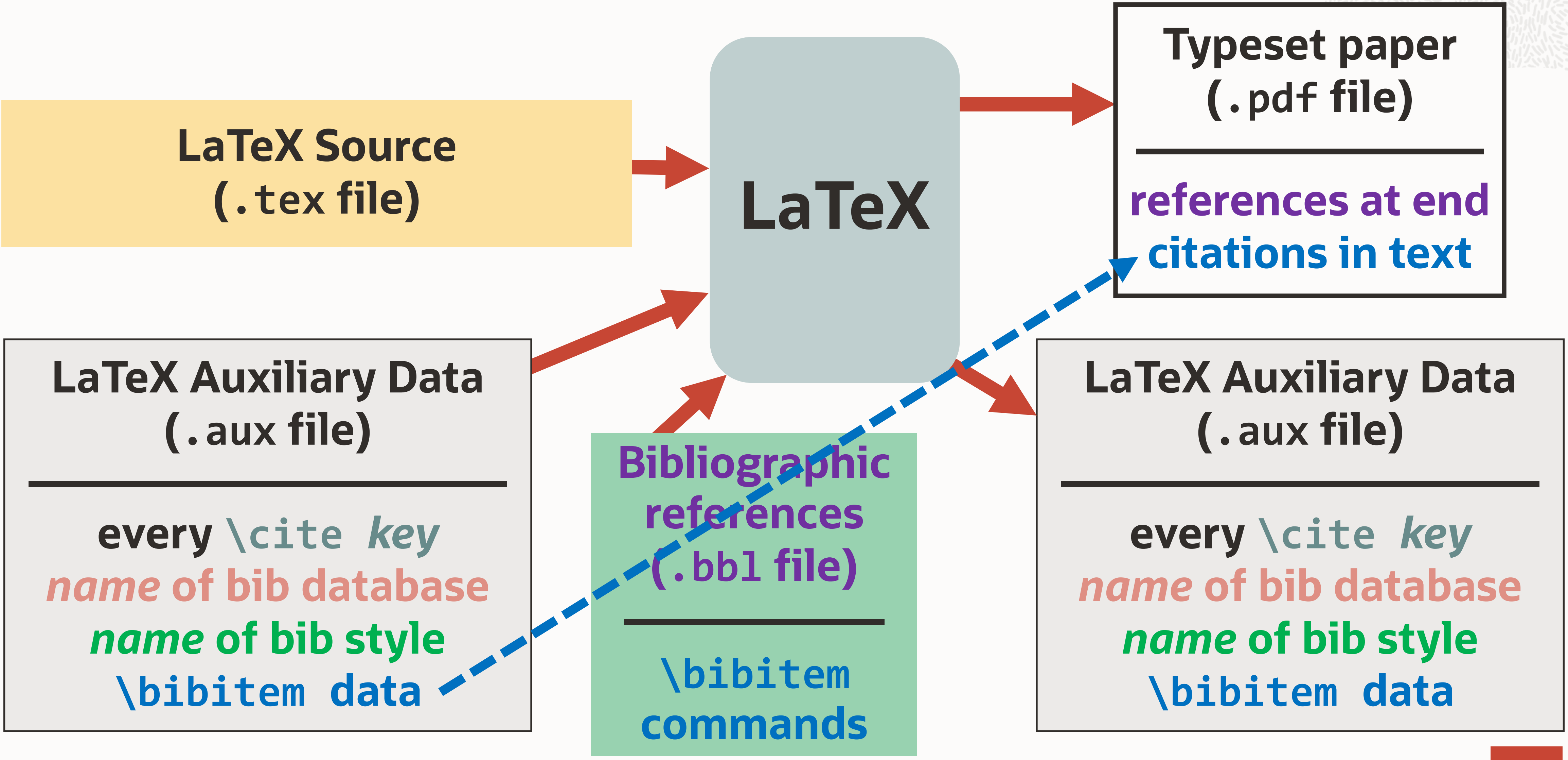
How BibTeX Works: Run BibTeX



How BibTeX Works: Second pass through LaTeX



How BibTeX Works: Third pass through LaTeX



The Bibliography Database File (.bib)

Bibliographic database (.bib file)

```
@preamble{"\newcommand\na{{\sc non-archival}}"}
@string{CACM = {Communications of the ACM}}
@string{oct = {October}}
@string{ACM = {Association for Computing Machinery}}
@string{NYC = {New York, NY, USA}}
@book{BLISS-Compiler,
  title = {The Design of an Optimizing Compiler},
  author = {William Wulf and Richard K. Johnson and
            Charles B. Weinstock and Steven O. Hobbs
            and Charles M. Geschke},
  ISBN = {0-444-00164-6},
  year = {1975},
  publisher = {American Elsevier},
  address = {New York}
}
@article{counters, author = {Morris, Robert},
title = {Counting Large Numbers of Events in Small Registers},
year = {1978}, journal = CACM, volume = {21}, number = {10},
doi = {10.1145/359619.359627}, month = oct, pages = {840-842}}
```

In the .tex file:

```
\cite{counters}

\cite{BLISS-Compiler}
```

LaTeX Input and PDF Output with Style angew

```
\documentclass{article}
\usepackage{natbib}
\usepackage{hyperref}

\bibliographystyle{angew}

\begin{document}
\noindent
We used the Bliss compiler
\citep{BLISS-Compiler} to
compile our implementation
of approximate counters
\citep{counters}.

\bibliography{test}
\end{document}
```

We used the Bliss compiler (Wulf *et al.*, 1975) to compile our implementation of approximate counters (Morris, 1978).

References

W. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobbs, C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, **1975**.

R. Morris, *Communications of the ACM* **1978**, *21*, 840–842.

LaTeX Input and PDF Output with Style erae

```
\documentclass{article}
\usepackage{natbib}
\usepackage{hyperref}

\bibliographystyle{erae}

\begin{document}
\noindent
We used the Bliss compiler
\citep{BLISS-Compiler} to
compile our implementation
of approximate counters
\citep{counters}.

\bibliography{test}
\end{document}
```

We used the Bliss compiler (Wulf et al., 1975) to compile our implementation of approximate counters (Morris, 1978).

References

Morris, R. (1978). Counting large numbers of events in small registers. *Communications of the ACM* 21: 840–842, doi:10.1145/359619.359627.

Wulf, W., Johnson, R. K., Weinstock, C. B., Hobbs, S. O. and Geschke, C. M. (1975). *The Design of an Optimizing Compiler*. New York: American Elsevier.

LaTeX Input and PDF Output with Style natdin

```
\documentclass{article}
\usepackage{natbib}
\usepackage{hyperref}

\bibliographystyle{natdin}

\begin{document}
\noindent
We used the Bliss compiler
\citep{BLISS-Compiler} to
compile our implementation
of approximate counters
\citep{counters}.

\bibliography{test}
\end{document}
```

We used the Bliss compiler (Wulf u. a., 1975) to compile our implementation of approximate counters (Morris, 1978).

References

[Morris 1978] MORRIS, Robert: Counting Large Numbers of Events in Small Registers. In: *Communications of the ACM* 21 (1978), October, Nr. 10, S. 840–842. `http://dx.doi.org/10.1145/359619.359627`. – DOI 10.1145/359619.359627

[Wulf u. a. 1975] WULF, William ; JOHNSON, Richard K. ; WEINSTOCK, Charles B. ; HOBBS, Steven O. ; GESCHKE, Charles M.: *The Design of an Optimizing Compiler*. New York : American Elsevier, 1975. – ISBN 0–444–00164–6

LaTeX Input and PDF Output with Style `plainnat`

```
\documentclass{article}
\usepackage[numbers]{natbib}
\usepackage{hyperref}

\bibliographystyle{plainnat}

\begin{document}
\noindent
We used the Bliss compiler
\citep{BLISS-Compiler} to
compile our implementation
of approximate counters
\citep{counters}.

\bibliography{test}
\end{document}
```

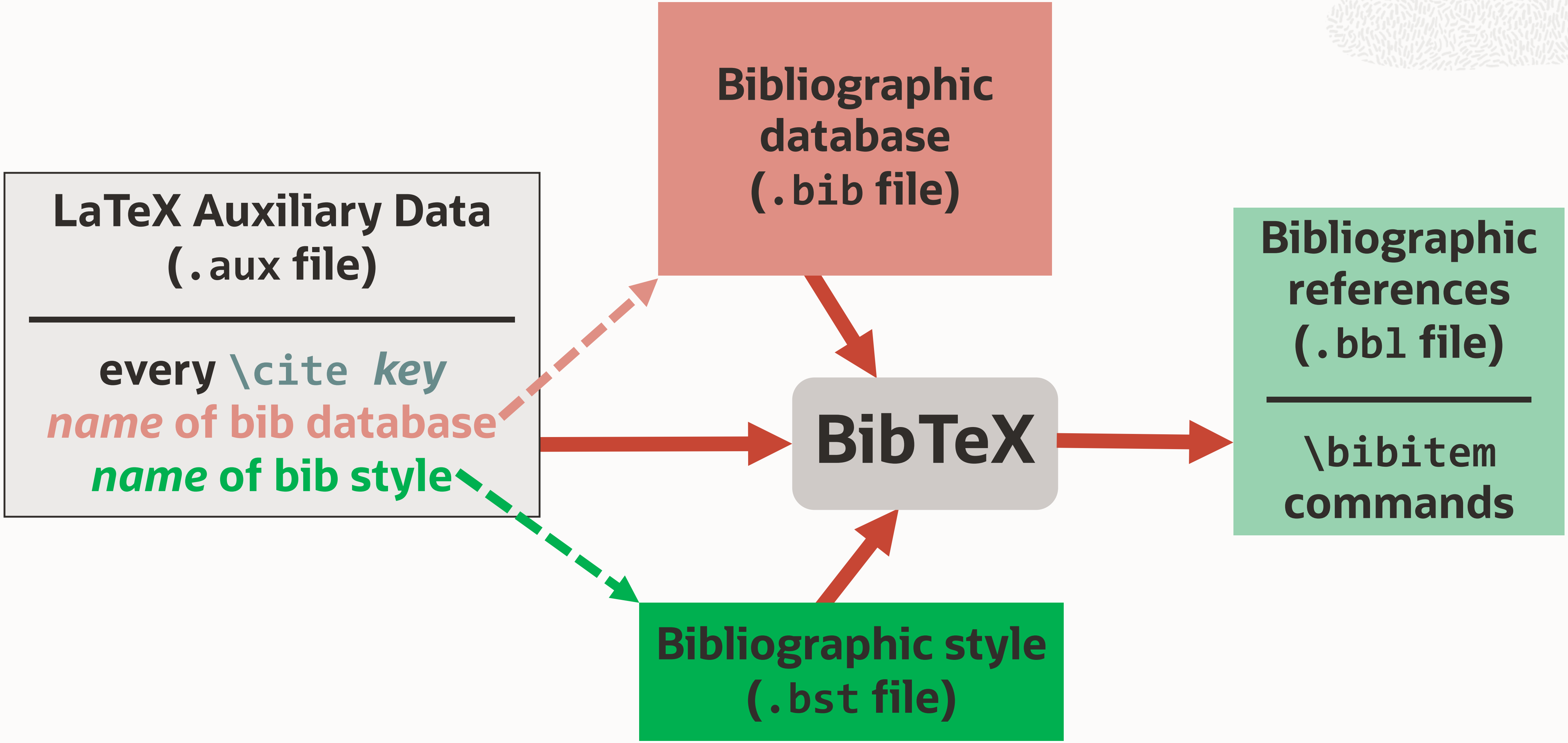
We used the Bliss compiler [2] to compile our implementation of approximate counters [1].

References

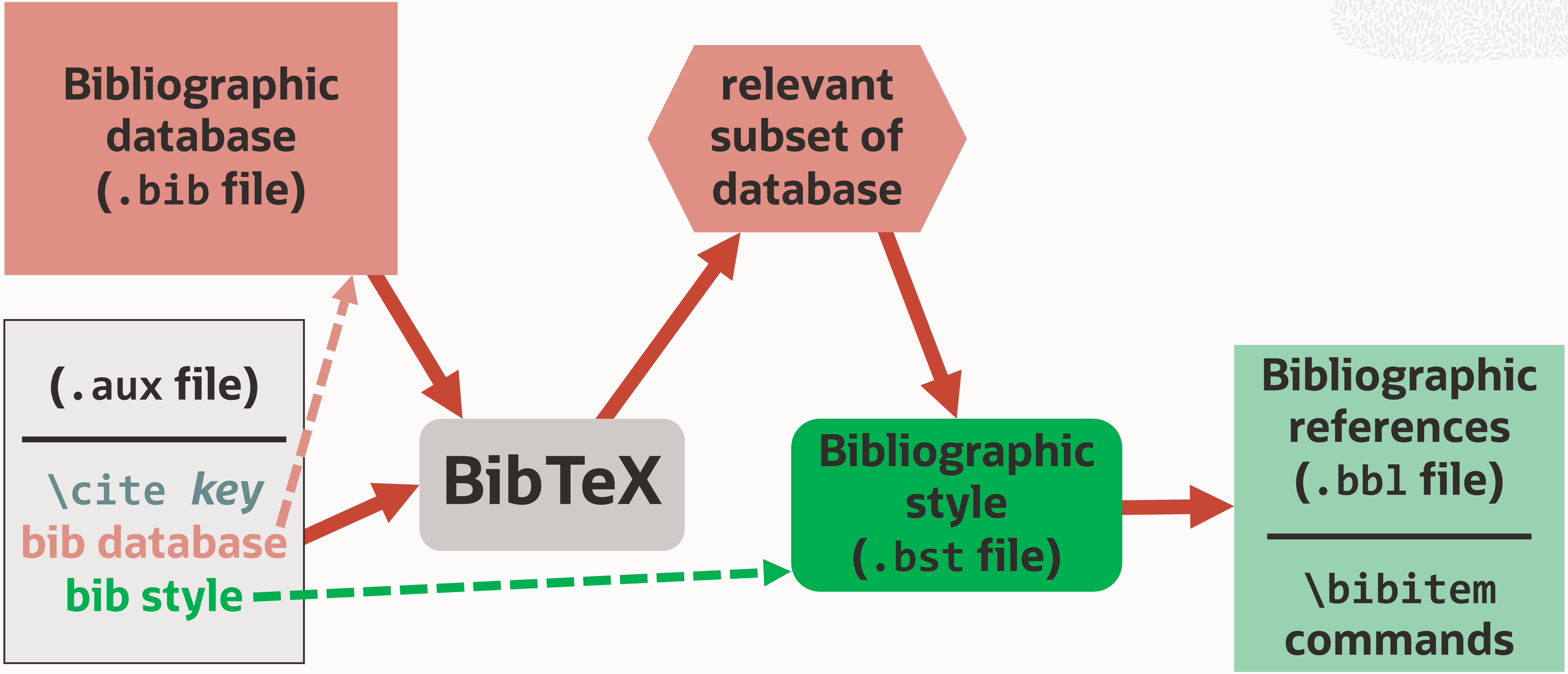
[1] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, October 1978. doi: 10.1145/359619.359627.

[2] William Wulf, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler*. American Elsevier, New York, 1975. ISBN 0-444-00164-6.

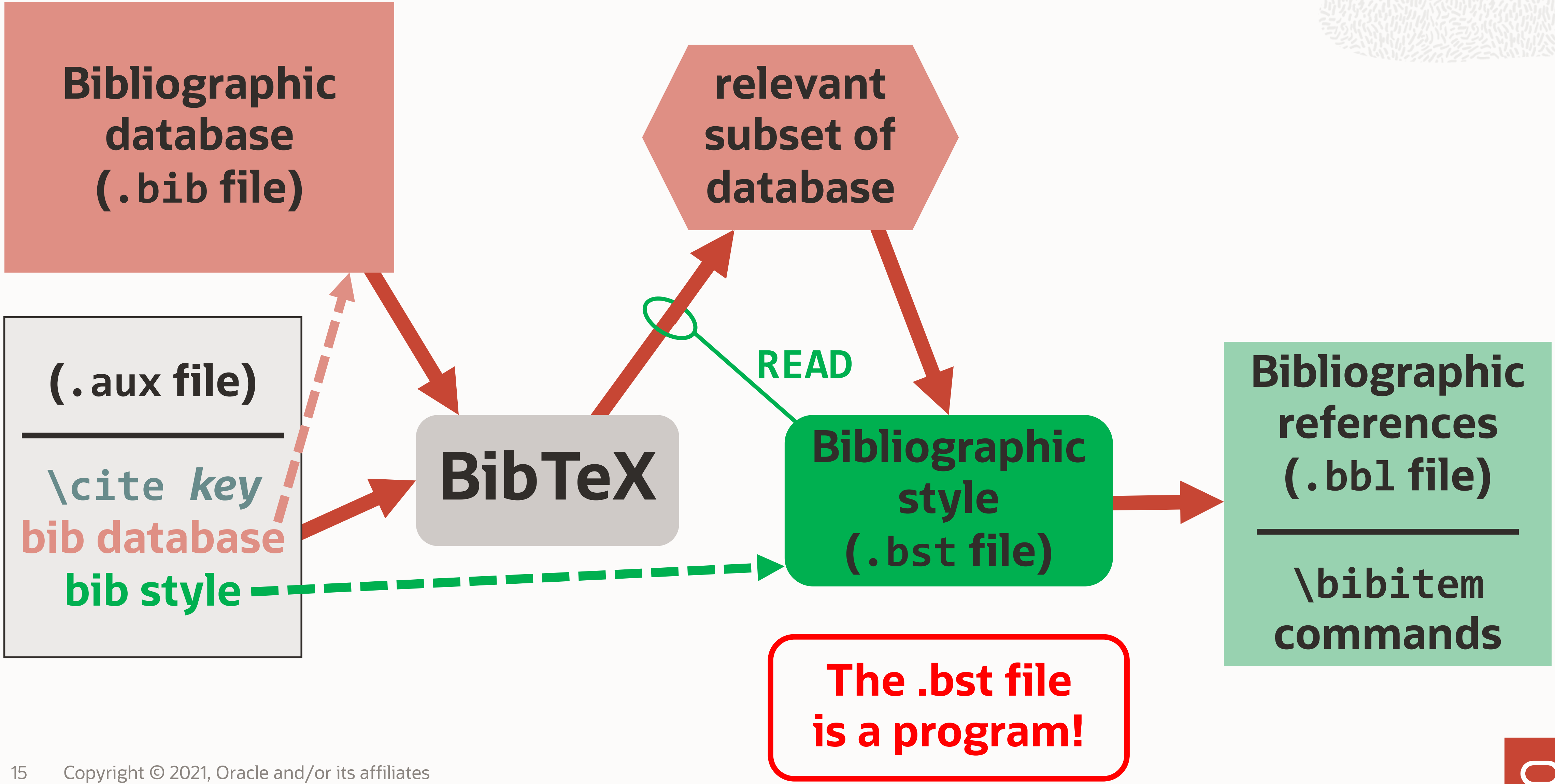
How BibTeX Works



How BibTeX "Really" Works



How BibTeX "Really" Works



The BibTeX Style Language

- Created by Oren Patashnik and Leslie Lamport in 1985.
- Version 0.98f released in March 1985.
- Version 0.99c released in February 1988.
- In 2003, publication of "BibTeX yesterday, today, and tomorrow"
  - Proposed 19 sets of changes to the language (never done)
- Version 0.99d released in March 2010 to improve the printing of URLs.

A language of the 1980s that has changed hardly at all for 33 years.

A solid workhorse used every day around the world.

More sophisticated replacements exist but have not displaced it.

Data Types in a .bst Program

- strings of ASCII (7-bit) characters
- integers (probably 32 bits? originally 16 bits!)
- functions
- empty (value of a database entry field for which no value was supplied)

The Data Environment for a .bst Program

1. Named things (one namespace)!

Functions

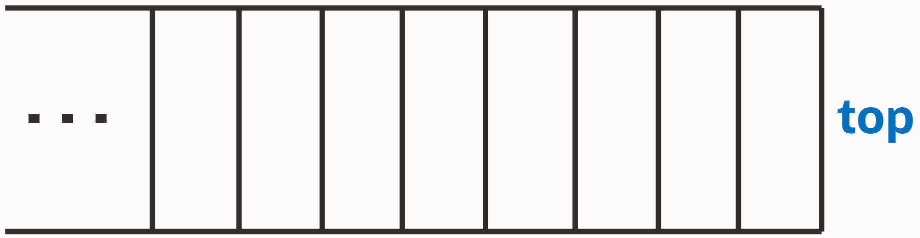
Macros

Global variables
- string variables
- integer variables

Database entry variables
- fields (string or empty)
- per-entry string variables
- per-entry integer variables

2. The (unnamed) list of chosen database entries

3. The stack



Each stack slot can contain one of:
- string
- integer
- function
- empty

Primitive functions take arguments on the stack and return results there.

Top-level Program Commands (1 of 6)

Declare database fields and entry variables:

ENTRY
  { *field-names* }
  { *per-entry-integer-variables* }
  { *per-entry-string-variables* }

```
ENTRY
  { author title journal volume number year month day
    pages publisher address note }
  { citation.order }
  { sort.year sort.label }
```

Top-level Program Commands (2 of 6)

Declare global variables and macros:

INTEGERS { *integer-variable-names* }
STRINGS { *string-variable-names* }
MACRO { *name* } { *string-literal* }

```
INTEGERS { numnames count len show-isbn-10-and-13 }
STRINGS { s t last.label }
MACRO { jan } { "January" }
```

`@string` in the `.bib` file can override a `MACRO` definition in the `.bst` file.

Top-level Program Commands (3 of 6)

Declare functions:

FUNCTION { *name* } { *list-of-functions-to-call* }

```
FUNCTION { double } { duplicate$ + }
FUNCTION { not }
{
    { #0 }
    { #1 }
  if$
}
FUNCTION { increment.count } { count #1 + 'count := }
```

It is forbidden for functions to be recursive.

Top-level Program Commands (4 of 6)

Execute a function from the top level:

EXECUTE { *function-name* }

Database entry variables are not available to the called function—only global variables.

Top-level Program Commands (5 of 6)

Work with the list of database entries:

`READ`	read `.bib` file and construct the list of database entries (the relevant subset of the full `.bib` database)
`SORT`	sort the list of database entries using the special implicit per-entry field `sort.key$`

Top-level Program Commands (6 of 6)

Execute a function once for each database entry:

```
ITERATE { function-name }

REVERSE { function-name }
```

Entry variables are available to the called function.

Typical Structure of a `.bst` File

One `ENTRY` command to define the database structure.
A mix of `STRINGS`, `INTEGERS`, `FUNCTION`, and `MACRO` declarations.
One `READ` command to set up the list of referenced entries
 (`MACRO/@string` references in the `.bib` file are processed at this time).
A mix of `EXECUTE`, `ITERATE`, `REVERSE`, and `SORT` commands
 (and possibly more `STRINGS`, `INTEGERS`, and `FUNCTION` declarations).

The last four lines are almost always something like:

```
ITERATE { call.type$ }
FUNCTION { end.bib }
  { newline$ "\end{thebibliography}" write$ newline$}
EXECUTE { end.bib }
```

Primitive Functions (1 of 11)

Functions take their arguments on the stack and return results there.

Simple arithmetic functions:

=	compare two integers: 1 if true, 0 if false
<	compare two integers: 1 if true, 0 if false
>	compare two integers: 1 if true, 0 if false
+	add two integers, leaves sum on stack
-	subtract two integers, leaves difference on stack

Simple string functions:

=	compare two strings: 1 if true, 0 if false
*	concatenate two strings, leaves result on stack

```
FUNCTION { times10 } { double duplicate$ double double + }
```

Primitive Functions (2 of 11)

Stack manipulation:

`duplicate$`	push a copy of top stack item
`swap$`	pop top two stack items, push back in other order
`pop$`	pop(and discard) top stack item

Type testing:

`missing$`	1 if top of stack is the value of a missing field, else 0
`empty$`	1 if top of stack is the value of a missing field or a string containing no non-whitespace characters, else 0

Primitive Functions (3 of 11)

Pushing onto stack:

#nnnn	integer literal: push integer onto stack
"*xxxx*"	string literal: push string onto stack
{ *list-of-functions* }	function literal: push function onto stack
variable-name	push value of the variable
field-name	push value of field, or empty if no value given
'*function*	push function onto stack
'*variable-name*	push the "variable-value push function"
'*field-name*	push the "field-value push function"
quote$	push string containing one double-quote character

***Could have defined that last one as*:**

```
FUNCTION { quote$ } { #34 int.to.chr$ }
```

Examples

```
FUNCTION { parenthesize } { "(" swap$ * ")" * }

FUNCTION { italicize } { "\emph{" swap$ * "}" * }

FUNCTION { format.title } { title italicize add.period$ }
```

Primitive Functions (4 of 11)

Assignment:

`:=` stack has a value and a push-function for a variable or field; pop them and assign the value to to the variable or field (signal an error if the value has the wrong type)

Control structure:

`if$` stack has "*integer*, *function1*, *function2*"; pop them, then call *function1* if *integer* is positive, otherwise call *function2*

`while$` stack has "*function1*, *function2*"; pop them; call *function1*, pop top of stack (must be an integer), and if value is positive, call *function2* and repeat

`skip$` do nothing

Examples

```
FUNCTION { format.title } {
  title empty.or.unknown
    { "" }
    { title italicize
      titleaddon empty.or.unknown
        'skip$
         { " " * titleaddon parenthesize * }
      if$
      add.period$
    }
  if$
}
```


Primitive Functions (5 of 11)

String operations:

`substring$`	**"*str*, *start*, *len*": compute ASCII-character substring (1-based indexing; if *start* is negative, count from end and *len* extends backward; tolerant of overshoot)**
`text.prefix$`	**"*str*, *len*": push a string containing the first *len* text characters of *str* ("`{\hat{o}}`" counts as one text character)**
`text.length$`	**"*str*": number of text characters in *str***

But there is no function that gives the ASCII-character length of a string!

Examples

```
FUNCTION { string.length } {
  #0 swap$
  { duplicate$ "" = not }
    { #2 global.max$ substring$ swap$ #1 + swap$ }
  while$
  pop$
}
```

It's slower than if it were a primitive, but that's okay; it turns out it's not needed that much in practice.

Primitive Functions (6 of 11)

Type conversions:

`int.to.chr$`	"*int*": convert ASCII value to single-character string
`chr.to.int$`	"*str*": convert single-character string to ASCII value
`int.to.str$`	"*int*": convert to signed-decimal representation

But there is no function `str.to.int$`.

Primitive Functions (7 of 11)

String operations:

`change.case$`	"*str*, *kind*": change case of letters in *str* (*kind* says how)
`add.period$`	"*str*": append a period to *str* unless the last non-"`}`" character is already "`.`" or "`?`" or "`!`"
`width$`	"*str*": physical width of *str* in hundredths of a point if typeset in the June 1987 version of font CMR10
`purify$`	"*str*": remove from *str* all characters other than letters, numbers, whitespace and hyphens and ties (which are turned into space characters), and certain other cases

Examples

```
STRINGS { s t }

FUNCTION { convert.to.lowercase } {
  's :=   "" 't :=
  { s "" = not }
    { s #1 #1 substring$ chr.to.int$
      s #2 global.max$ substring$ 's :=
      duplicate$ duplicate$
      "A" chr.to.int$ < not swap$ "Z" chr.to.int$ > not and
        { "A" chr.to.int$ - "a" chr.to.int$ + }
        'skip$
      if$
      int.to.chr$ t swap$ * 't :=
    }
  while$
  t
}
```

Primitive Functions (8 of 11)

Name-formatting operations:

These operate on a string of the form "*name* and *name* and … and *name*".

`num.names$`	**"*str*": return number of names in the string**
`format.name$`	**"*str*, *k*, *fmt*": (1) extract the *k*'th name from *str* (2) decompose it into "*first*, *von*, *last*, *jr*" parts (3) reassemble these parts according to format string *fmt***

This works for many names, but not 100%.

Can We Call a Function?

There is no operation that, given a function on the stack, calls it.

But we can define one (and even name it so that it appears to be primitive):

```
FUNCTION { call$ } { #1 swap$ duplicate$ if$ }
```

Mapping a Function over a String

Example: "abc" 'parenthesize map *produces* "(a)(b)(c)"

```
STRINGS { s t }
FUNCTION { map } {                    % stack in: ... str fn
  swap$ 's :=   "" 't :=
  { s "" = not }
    { duplicate$
      s #1 #1 substring$ chr.to.int$
      s #2 global.max$ substring$ 's :=
      swap$ call$
      t swap$ * 't :=
    }
  while$
  pop$
  t
}
```

Can We Define the S, K, I Combinators?

$$I\ z \Rightarrow z$$

$$K\ x\ z \Rightarrow x$$

$$S\ x\ y\ z \Rightarrow (x\ z)\ (y\ z)$$

```
FUNCTION { I } { skip$ }
FUNCTION { K } { pop$ }
FUNCTION { S } { 'z :=  z swap$ call$ swap$ z swap$ call$ call$ }
```

But there is a problem: *z* might be a function.

- **Can't put *z* in a global variable.**
- **Can't get at the third thing down on the stack (only have `swap$`).**

There Is Actually a Deeper Problem

$$I\ z \Longrightarrow z$$

$$K\ x\ z \Longrightarrow x$$

$$S\ x\ y\ z \Longrightarrow (x\ z)\ (y\ z)$$

When we use the combinatory calculus as a target language for translating the lambda calculus, we depend on being able to *curry* `K` and `S`.

We need to be able to call `K` with one argument and get back a function.

But this doesn't work:

```
FUNCTION { K } { 'x :=  { pop$ x } }
```

In this language, all variables are global, and functions don't have environments.

There is no way to return a function that remembers any calculated values.

So maybe this isn't really a functional language (in the usual sense) after all.

Primitive Functions (9 of 11)

Functions on the database or current database entry:

`cite$`	push a string for the citation key of the current entry (used as the argument to `\cite`)
`type$`	push a lowercase string for the type of the current entry (`"book"`, `"article"`, `"inproceedings"`, etc.) or an empty string if the type is unknown (there is no function with that name)
`call.type$`	for the current entry, if `type$` is not an empty string, call the function that has that name; otherwise call the function `default.type`
`preamble$`	push a string that is the concatenation of all arguments to the `@preamble` command in the `.bib` file

Primitive Functions (10 of 11)

Writing to the `.bbl` file:

`write$`	pop top stack item (must be a string) and write it to the `.bbl` file
`newline$`	write a newline to the `.bbl` file

Primitive Functions (11 of 11)

Error reporting and debugging:

`warning$` pop top stack item (must be a string), prepend `"Warning--"`, print it, and increment count of warnings

`top$` pop and print top stack item on terminal (for debugging)

Feature: you can drop in `"Reached point A" top$` *anywhere*.

Problem: `"34"` *and* `#34` *print the same (just the two digits), and you can't write an improved version because the language has no way to test the type of a stack item.*

`stack$` pop and print *all* stack items on terminal (for debugging)

Problem: it pops the stack, so you can't continue execution.

My Project: Update `ACM-Reference-Format.bst` for HOPL IV Conference

Task: implement dates in ISO 8601 format (`yyyy-mm-ddThh:mm:ss±zh:zm`)

- **Provide complete timestamps for email and social media**
- **Avoid splitting time and date info across multiple fields**
  - **Example: `date = {2019-10-02}` rather than `year`, `month`, `day` fields**
- **Want to map numeric month values to text**

Problem: macros are inaccessible to the programmer
Workaround: duplicate this information in the program

Problem: `@string` overrides in the `.bst` file will not be used for this purpose
Workaround:

- **Extend ISO 8601 support so that `date = {2019January02}` also works**
- **In `.bib` file, actually write `date = {2020} # jan # {02},`**

Language extension? Allow `jan` in a program to push the macro string

My Project: Update `ACM-Reference-Format.bst` for HOPL IV Conference

Task: better support East Asian, Spanish, and Jr.-without-comma names

Problem: `format.name$` **has a specific, built-in theory of name parts and is not extensible.**

Solution: Completely reimplement `format.name$` **in the style language!**
(I actually haven't finished this part of the project yet.)

Problem: The code that implements `format.name$` **is quite complex (about 1/6 of all of BibTeX!).**

It would have been easier if the three things that `format.name$` **does were three separate primitive functions:**

n `select.name$` `name.parts$` *fmt-string* `format.name.parts$`

Then I could just replace the `name.parts$` **function, not the whole thing.**

A Few Observations (1 of 6)

A primitive that is too specific may be ignored or completely reimplemented—and then what good is it?

`add.period$` could have handled more cases.

`add.period$` could have handled adding a comma.

`format.names$` doesn't handle certain names properly and suggested workarounds don't work for all `.bst` files.

A Few Observations (2 of 6)

If a primitive does several independent things (for example, `format.name$`, `stack$`, `warning$`), splitting those things apart can provide future flexibility.

On the other hand, there is value in demonstrating the right way to combine them for a specific task.

One can provide either an extra primitive or a library function.

The whole question of whether a feature should be a primitive or a library function can be a difficult design decision.

(Observation: the BibTeX style language does not support libraries well. Having even a simple `INCLUDE` command would help a lot.)

A Few Observations (3 of 6)

In a stack language, *deconstructors* such as `name.parts$` can be valuable.
(In an expression-based language, this may show up as *pattern matching*.)

```
FUNCTION { head.tail }
{ duplicate$ #1 #1 substring$ swap$ #2 global.max$ substring$ }
```

Yet another example: splitting an ISO 8601 date into many pieces.

Stack before: . . . *str*
Stack after: . . . *year month day hour minute second timezone season*

A Few Observations (4 of 6)

Sometimes it is a good idea to provide a complete (or expected, or symmetric) set of operations on a well-known data type, even if you think not all the operations will be used in practice. This provides future flexibility and can help prevent misuse.

For integers: `multiply`, `divide`, `and`, `or`, `xor`, `shift`?

For strings: `find`, `split`? Both kinds of `length`?

For functions: `call`, `map`, `mapreduce`? Function-valued variables?

If not, then at least think carefully about how such operations can/will be programmed in terms of existing primitives (and what mistakes might be made).

A Few Observations (5 of 6)

Sometimes it is a good idea to provide primitives to probe and affect the programming environment.

Find out the type of a stack item.

Access the size of the stack; access any stack slot without popping.

Access the values of macro names.

Find out all names of all fields in an entry, even if not declared.

If nothing else, this may allow the construction of better error reporting and debugging tools.

A Few Observations (6 of 6)

Sometimes the application domain shifts, and you need a major overhaul.

Support UTF-8, not just ASCII.

This would actually fit in well with the existing concept of text characters.

Final Observations

A Domain-Specific Language does need to address its specific domain.
But it also needs to "be a language".

It needs features that support the specific domain.
But it also needs features (or tools) that support "being a language".

There are design tradeoffs between doing one specific thing well and leaving room for future programming of variations.

The domain may change; can the language change with it?

- ***If so, can these changes be made from within the language?***
  - ***If so, does the language design make such changes easy or hard?***