A Tour around the $\mathcal{NTS}$ implementation

Karel Skoupý

Reasons for Making a New System

- new functionality was needed
- interoperability with other systems and formats was needed

Why not just extend TEX?

- TEX code is difficult to understand
  design is clean but abstraction is missing
- TEX code is difficult to extend
  many global variables, dependencies, overused data structures

Objectives of the Reimplementation

- system which behaves exactly like $\TeX$
- code which is easy to understand
- code which is easy to modify/extend
- components that can be reused in other systems

How to achieve that?

- modular structure
- clear module interfaces
- explicit inter-module dependencies
- as less dependencies as possible
- high level of abstraction

The Expectations

- $\mathcal{NTS}$ will be simple, everybody can understand it
- everybody will be able to take it and modify it
- all problems will be magically solved

And the reality?

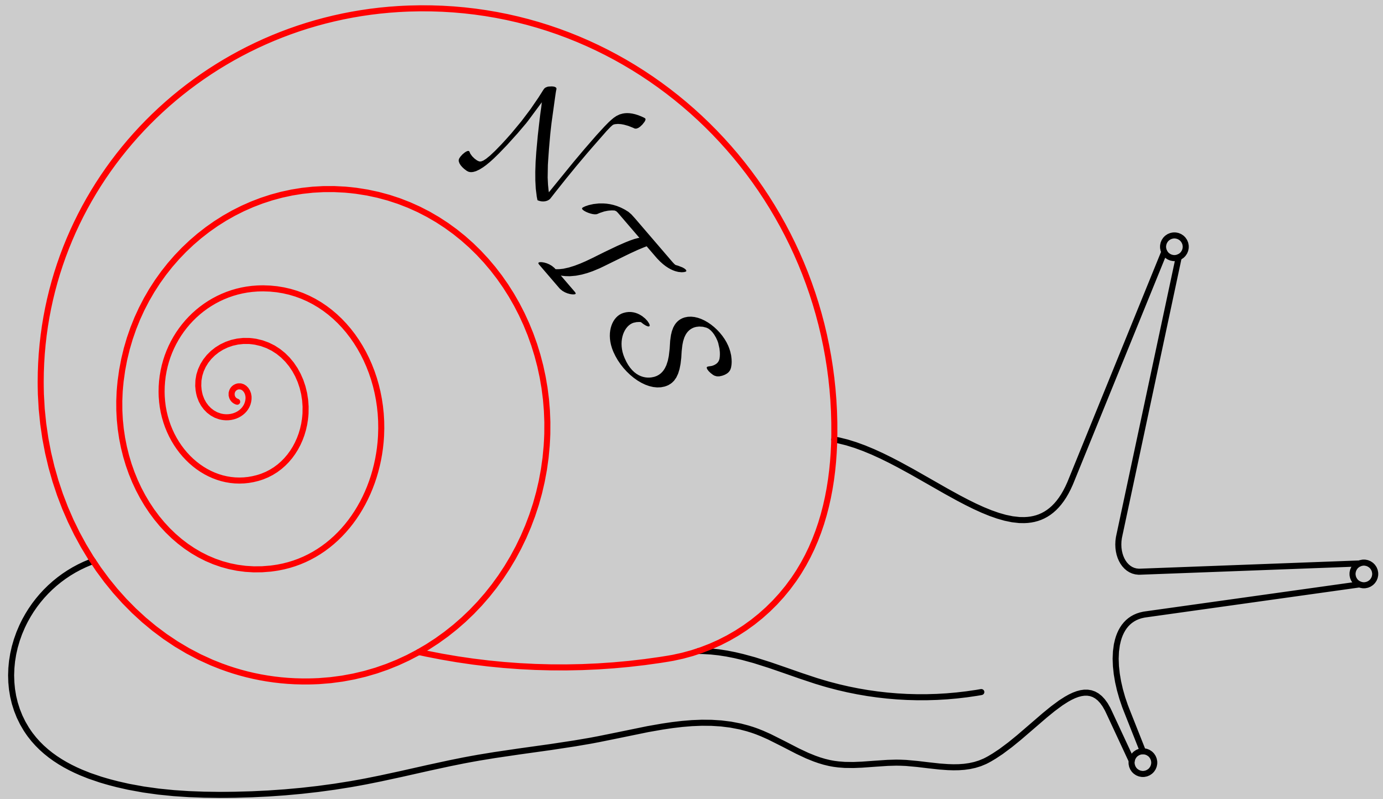
- $\mathcal{NTS}$ performs the same processing as TEX does, not simpler
- the code is sometimes even more complex because quick hacks are not allowed
- there is 13 packages, 532 source files, 641 named classes, and 82 interfaces
- aproximately twice as much lines of code as in TeX

Look Closer

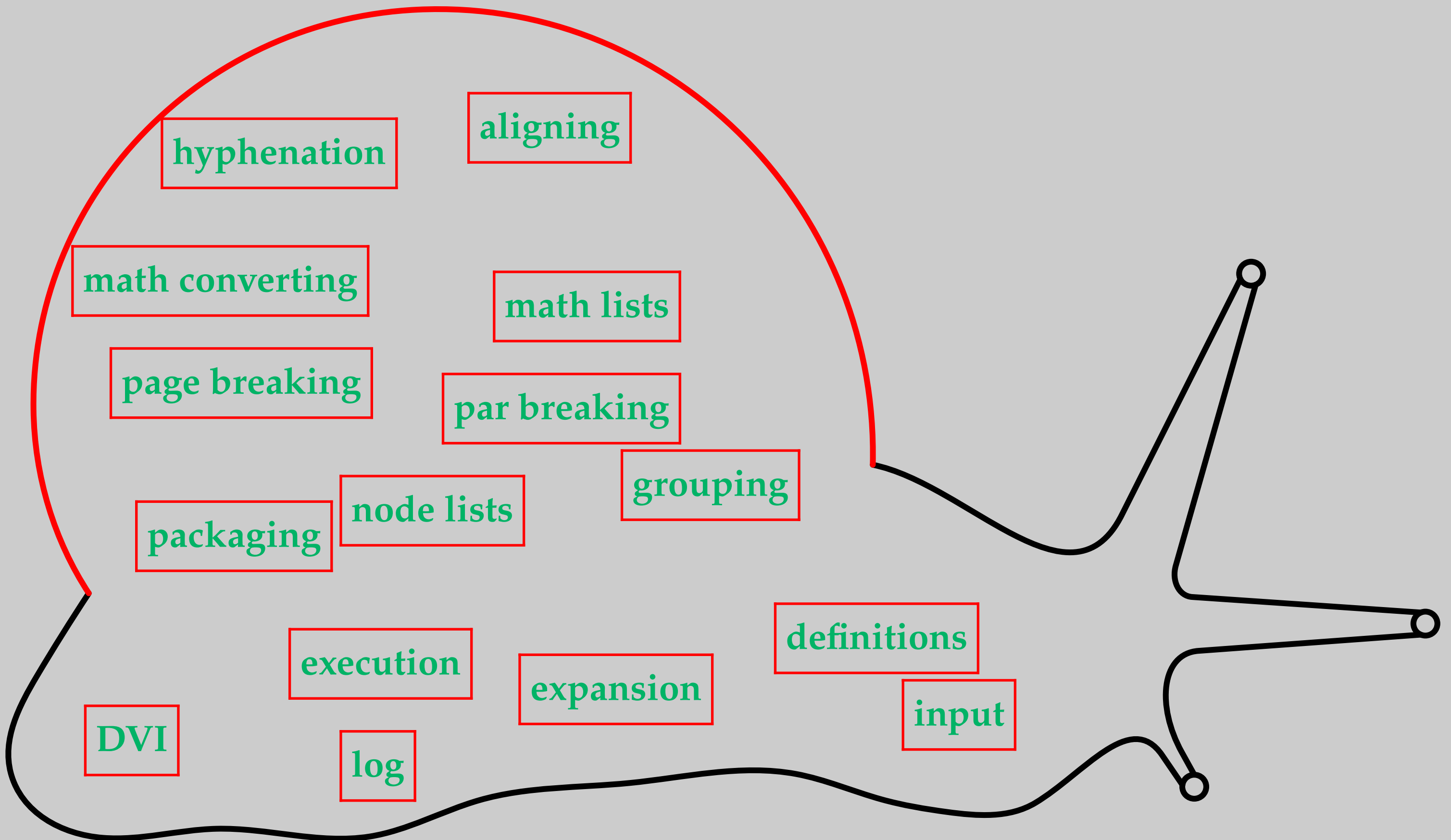
- there is only about 50 key concepts
- clear interfaces are defined
- implementation details are hidden inside classes
- there is strict dependency hierarchy of packages
- source files are very short in most cases

Distribution of source file sizes

Anatomy of



The Anatomy



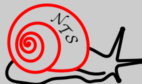
Line Input and Tokenization

Class `nts.io.LineInput`

- reads an input line by line
  `readLine`
- returns `InputLine`s which consist of `CharCode`s

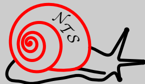
Class `nts.command.InputLineTokenizer`

- provides stream of `Tokens` from line of `CharCodes`
  `nextToken`
- is parametrized by `TokenMaker` which knows the catcodes



Interface `nts.tex.FileOpener`

- defines abstract methods for opening files
  `openForReading`, `openForWriting`



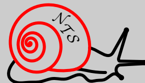
Expanding `Tokens`, evaluating conditionals

Class `nts.command.Token`

- has always some associated `Command`
  `meaning`
- its meaning is checked whether it is expandable
  `expandable`, `doExpansion`

Class `nts.command.Expandable`

- common ancestor for most of expandable `Command`s
- usually pushes some `Tokenizer` to `TokenizerStack`
  `Macro.Expansion`

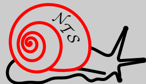


Class `nts.command.CommandBase`

- maintains a `TokenizerStack`
  `getTokStack`
- provides methods for getting and pushing `Token`s
  `nextRawToken`, `nextExpToken`, `pushToken`, `pushList`

Class `nts.command.CondPrim`

- maintains internal stack of branches
  which have to be accepted or skipped
- provides common features for specialized conditionals
  `IfBoolPrim`, `IfCasePrim`



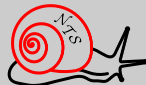
Executing `Command`s

Class `nts.command.Command`

- defines abstract interface to every `Command`
  every `Command` knows ho to execute
  `exec`
- certain `Commands` can provide values in certain context
  `hasDimenValue`, `getGlueValue`

Interface `nts.command.Primitive`

- provides a name so it can be easily registered
  under that name as a command in `Primitives`



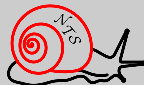
Class `nts.command.CommandBase`

- defines a lot of useful methods for scanning
  `nextRawToken`, `scanNum`, `scanDimen`, `scanGlue`
- for logging and error reporting
  `normLog`, `error`, `backToken`
- contains only static methods and is inherited by many classes
  `Command`, `Group`, `Action`

Assignments, definitions

Class `nts.base.LevelEqTable`

- stores mapping between keys and values
  `get`, `put`, `gput`
- maintains pushing and poping of levels
  `pushLevel`, `popLevel`
- saves values for external clients



Class `nts.command.AssignPrim`

- abstract ancestor of most assignable commands

- stores its value or values in `EqTable`
  `assign`, `getNumValue`

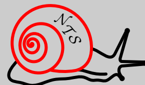
Class `nts.command.DefPrim`

- creates `Macro`s and associates them in `EqTable`
  as a meaning of `CtrlSeqToken`s or `ActiveCharToken`

Building `Node` lists

Class `nts.builder.Builder`

- maintains static stack of `Builders`
  `push`, `pop`, `top`
- appends new `Node`s to its internal list
  `addKern`, `addSkip`, `addPenalty`, `addRule`, `addBox`, `addNode`, `addNodes`
- returns the resulting `NodeList`
  `getList`



Class `nts.typo.BuilderCommand`

- performs `Action` specific to current `Builder` (mode)
  `exec`
- the association of `Action`s to `Command`s and `Builder`s
  is defined in `nts.tex.Primitives`

Class `nts.typo.TypoCommand`

- maintains the current `FontMetric`
  `getCurrFontMetric`, `setCurrFontMetric`
- contains general methods common to other typographic commands
  `appendChar`, `appendNormalSpace`, `packHbox`, `packVbox`
- uses `WordBuilder` provided by `FontMetric` for building ligatures and kerns

  `getWordBuilder`, `add`, `close`

Packaging `Node` lists

Class `nts.node.SizesIterator`

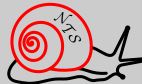
- provides sequence of abstract items
  `hasNextElement`, `takeNextElement`
- gives the relevant sizes of the current item
  `currWidth`, `currHeight`, `currDepth`
- there are implementation for both directions
  `HorizIterator`, `VertIterator`

Class `nts.node.SizesEvaluator`

- accumulates sizes of items
  `add`, `addShrink`, `addStretch`
- provides the resulting sizes and `GlueSetting`
  `getWidth`, `getHeight`, `getBody`, `getDepth`
  `getSetting`, `getBadness`

Class `nts.typo.TypoCommand`

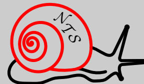
- handles the usual cases via its inner classes
  `HBoxPacker`, `VBoxPacker`
  `packHbox`, `packVbox`



Grouping

Class `nts.command.Group`

- stack of `Group`s is maintained by `CommandBase`
  `pushLevel`, `popLevel`
- declares methods which are called when pushed and poped
  `open`, `start`, `stop`, `close`
- checks which `Command`s can finish group
  `defineClosing`, `expectedToken`
- typographic groups push a `Builder` on beginning
  and pop and use it on finishing



Breaking paragraphs

Class `nts.node.Breaker`

- generic breaker which breaks given `NodeList`
  `breakToLines`
- returns sequence of broken lines
  `hasMoreLines`, `getNextLine`

Class `nts.typo.Paragraph`

- collects the current horizontal list and invokes `Breaker` `lineBreak`
- provides context to `Breaker` `ParBreaker`
- processes the resulting lines

Hyphenation

Class `nts.typo.HyphenNodeEnum`

- filters ordinary `Node` stream and contributes the `DiscretionaryNode`s
- is used by `Paragraph` if needed

Class `nts.hyph.WordTree`

- maintains a set of patterns
- provides information about possible breaks in a word

Breaking pages

Class `nts.node.VertSplit`

- generic breaker which breaks given `NodeList`
  `tryBreak`, `findBreak`

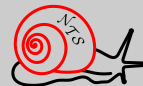
Class `nts.node.PageSplit`

- extends `VertSplit` and cares about `Insertions`

- checks if the page is already full
  `build`

Class `nts.typo.Page`

- provides context to `PageSplit`
- maintains the current vertical list
- performs output of the finished page
  `performOutput`



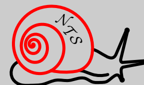
Building `Noad` lists

Class `nts.math.MathBuilder`

- appends new `Noad`s to its internal list
  `addKern`, `addSkip`, `addPenalty`, `addRule`, `addBox`,
  `addNoad`, `addNoads`, `addNode`, `addNodes`
- returns the resulting `NoadList`
  `getList`

Class `nts.math.MathPrim`

- contains general methods common to other math primitives
  `setMathChar`, `handleMathCode`
  `scanField`, `scanDelimiter`



Converting `Noad` lists into `Node` lists

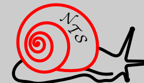
Interface `nts.noad.Noad`

- subclasses know how to convert themselves when a `Converter` is provided
  `convert`, `convertWithScripts`

- some subclasses contain `Field`s which know how to convert themselves as well

Interface `nts.noad.Converter`

- defines conversion context to `Noad`s and `Field`s
  `getStyle`, `getDimPar`, `fetchCharNode`

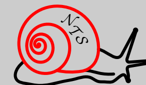


Class `nts.noad.Conversion`

- performs two-pass conversion of `NoadList`s
  `convert`
- provides context in form of `Converter` to `Noads` passed

Class `nts.math.FormulaGroup`

- invokes the conversion of finished math list
- supplies concrete style to conversion process



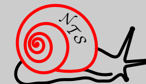
Converting `Node` lists into DVI

Interface `nts.node.TypeSetter`

- defines abstract typesetting methods
  `set`, `setRule`, `moveRight`, `startPage`
- every `Node` knows how to typeset itself by `TypeSetter`
  `typeSet`

Class `nts.dvi.DviTypeSetter`

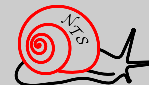
- is concrete `TypeSetter` for DVI format
- uses `DviFormatWriter` for low-level operations



Aligning

Class `nts.align.Alignment`

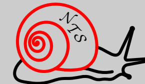
- scans and applies the alignment `Ppeamble`
  `scanPreamble`, `startColumnBody`
- maintains internal stack of `Alignment`s
- guards the rows and columns
  `finishColumn`
- transforms the cacumulated lists into tables
  `transform`



Class `nts.align.AlignPrim`

- invokes alignment mode

- there are subclasses for both directions
  `HAlignPrim`, `VAlignPrim`



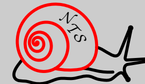
Log output

Interface `nts.io.Loggable`

- declares ability to write itself on `Log` output
  `addOn`
- majority of classes in $\mathcal{NTS}$ implements it

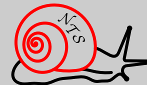
Interface `\nts.io.Log`

- defines methods to write basic types and `Loggable`
  `add`
- provides simple control over log output
  `startLine`, `endLine`



Class `nts.io.LineOutput`

- low level features for writing characters and lines
  `add`, `startLine`, `endLine`

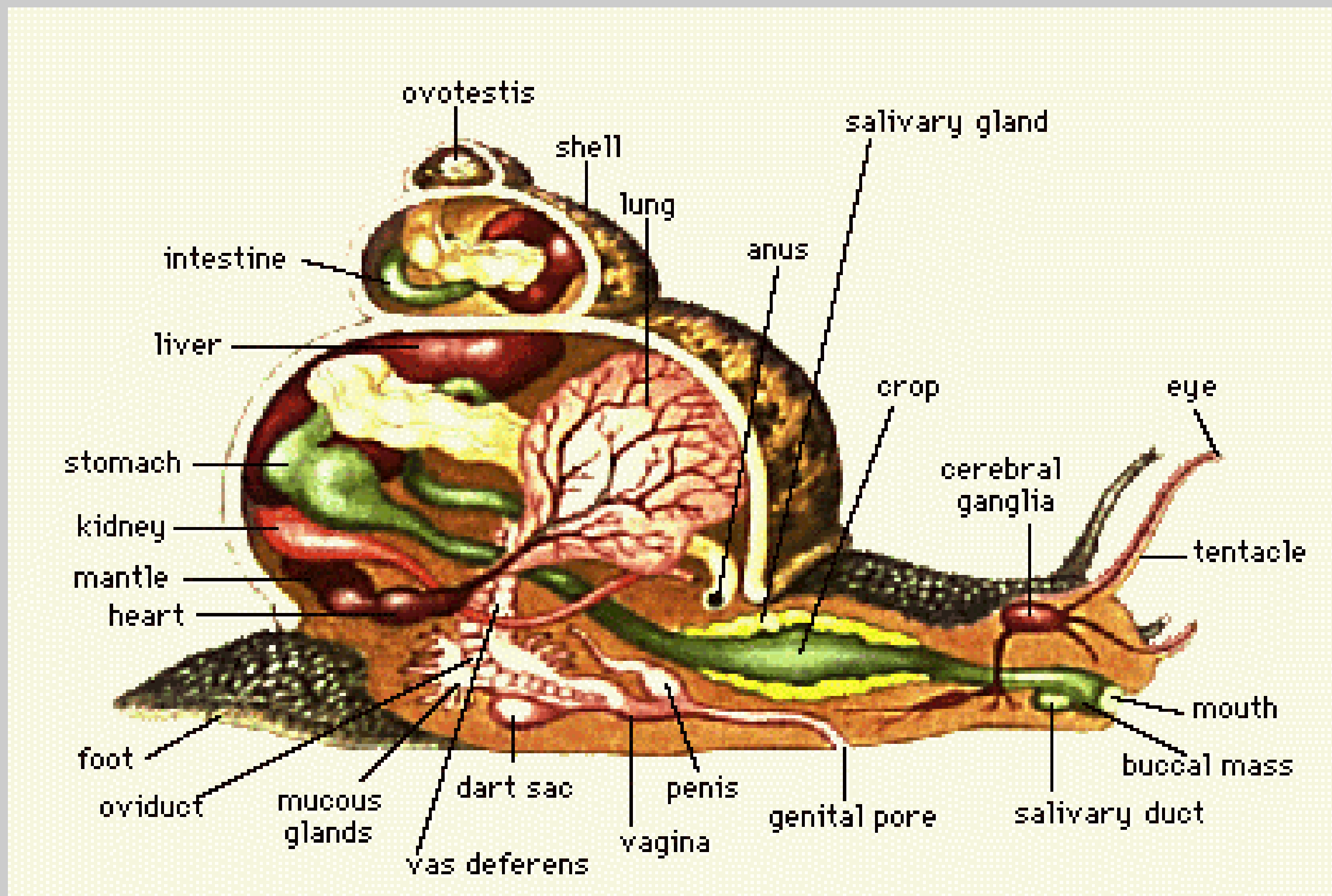


Digging Tips

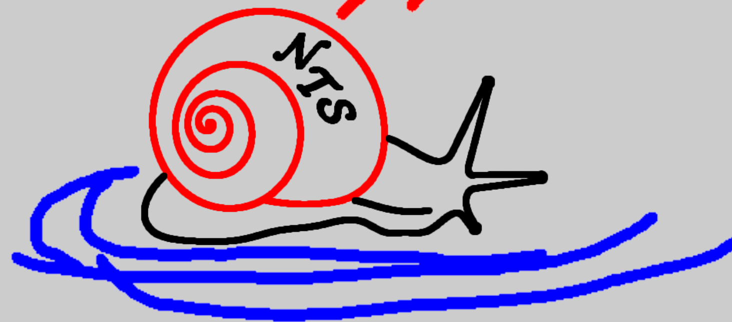
- generate tags; look up classes and methods which are used
- generate Java documentation; browse the class tree
- look for classes which are roots of subtrees
- look for names resembling familiar $\TeX$ concepts
- find out which source files are long

Future Plans

- finishing and conservation of $\mathcal{NTS}$
- starting another project with different style and priorities
- making the new system faster and more usable
- adding some new functionality
- working on harder problems; researching



Anatomy of a Real Snail



$\mathcal{NTS}$ in its Full Speed