UNIVERSITY OF CALIFORNIA, SAN DIEGO

Building Simple Annotation Tools

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Gordon Lin

Committee in charge:

Professor Chun-Nan Hsu, Chair
Professor Kamalika Chaudhuri, Co-Chair
Professor Ranjit Jhala

2016

EPIGRAPH

Fools ignore complexity. Pragmatists suffer it.

Some can avoid it. Geniuses remove it.

—Alan Perlis

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Professor Chun-Nan Hsu for his support and patience throughout my masters and thesis work.

I would also like to thank Professor Kamalika Chaudhuri and Professor Ranjit Jhala for taking the time to review this thesis.

ABSTRACT OF THE THESIS

Building Simple Annotation Tools

by

Gordon Lin

Master of Science in Computer Science

University of California, San Diego, 2016

Professor Chun-Nan Hsu, Chair
Professor Kamalika Chaudhuri, Co-Chair

The right annotation tool does not always exist for processing a particular natural language task. In these scenarios, researchers are required to build new annotation tools to fit the tasks at hand. However, developing new annotation tools is difficult and inefficient. There has not been careful consideration of software complexity in current annotation tools. Due to the problems of complexity, new annotation tools must reimplement common annotation features despite the availability of implementations in open sourced tools.

If new tools continue to ignore software complexity, their development will remain difficult and inefficient. This thesis advocates minimizing complexity for annotation tools in two ways: we raise awareness of complexity in annotation tools, and we propose our solution for assisting the development of simple annotation tools. We present our library, Notate, which provides simple implementations for building common features with in annotation tools.

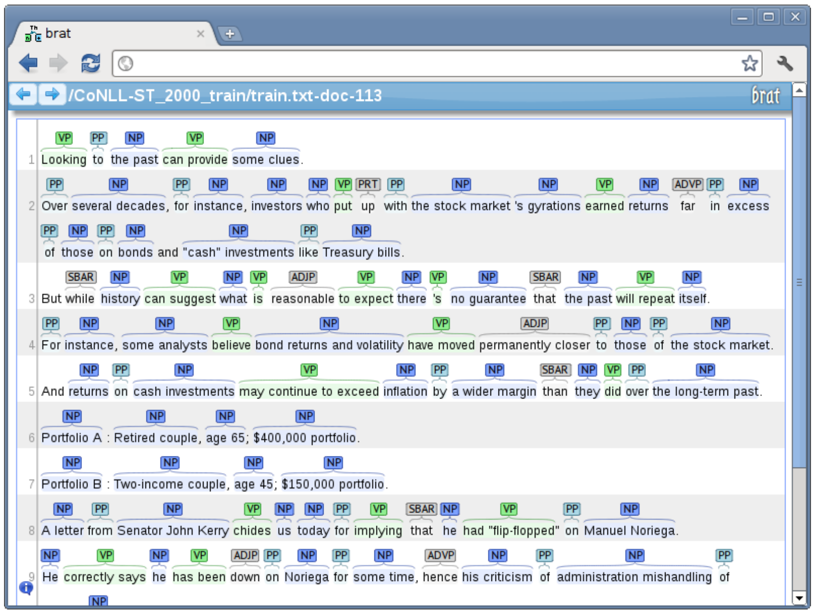
Chapter 1

Complexity of Annotation Tools

1.1 Introduction

Annotation tools are important components for many natural language processing (NLP) projects. They provide interfaces for visualizing and interacting with annotations within text documents, relieving annotators the tedious management of the underlying data format. There are a wide selection of tools available, each with their own strengths and weaknesses. It important for researchers to choose tools suitable for their project's tasks.

However, there are situations where no suitable tool exist, where all available tools lack functionality required for the desired task. In these cases, researchers will need to either create new tools or modify existing ones. Finlayson and Erjavec [FE16] suggest modifying existing tools over creating new ones, due to problems from implementing new tools. The three problems they mention are: reimplementation of behaviors in existing tools, early design errors stopping implementation down the line, and expensive maintenance cost from bug fixes and support requests. Finlayson and Erjavec do not address these problems and recommend modifying existing tools instead.



1 Looking to the past can provide some clues.
2 Over several decades, for instance, investors who put up with the stock market 's gyrations earned returns far in excess of those on bonds and "cash" investments like Treasury bills.
3 But while history can suggest what is reasonable to expect there 's no guarantee that the past will repeat itself.
4 For instance, some analysts believe bond returns and volatility have moved permanently closer to those of the stock market.
5 And returns on cash investments may continue to exceed inflation by a wider margin than they did over the long-term past.
6 Portfolio A : Retired couple, age 65; $400,000 portfolio.
7 Portfolio B : Two-income couple, age 45; $150,000 portfolio.
8 A letter from Senator John Kerry chides us today for implying that he had "flip-flopped" on Manuel Noriega.
9 He correctly says he has been down on Noriega for some time, hence his criticism of administration mishandling of

Figure 1.1: The brat rapid annotation tool (BRAT) [SPT+12] is popular for NLP projects and research.

However, modifying existing tools is not always feasible. Common problems such as poor documentation, lack of support, and software bugs [FE16] make existing tools expensive to change. The cost is increased if usability is a concern, as many tools exhibit usability violations [Bur12, Bur14]. In these cases, it is cheaper and faster to create new tools from scratch, especially if the tools only require a small set of specialized features. We see evidence of this in crowdsourcing studies where annotation tools have been incorporated into crowdsourced microtasks. Many crowdsourced annotation studies [GNWS14, YYSXH10, ZLD+13] have built their own custom annotation tools from scratch.

Initially, we were interested in exploring use cases for annotation tools on crowdsourcing platforms. Unfortunately, the existing tools were not easily reusable for crowdsourcing due to their inherent problems. We needed to build new annotation tools like many other annotation crowdsourcing studies. However, we realized that building new annotation tools is a significant challenge, and any naive attempts at their development will lead to problems alluded by Finlayson and Erjavec. Instead of building new tools, we would like to discuss and solve the issues which arise in their development. We believe these problems are caused by software complexity and understanding it will make developing annotation tools easier. In this chapter, we discuss why complexity is problematic, the benefits of simplicity, and sources of complexity in annotation tools.

1.2 Software Complexity

In this section, we argue the importance of understanding software complexity. We define what complexity means, why it is problematic, and the difficulty in removing it. We draw our arguments from Rich Hickey's *Simple Made Easy* [Hic11], Ben Moseley and Peter Marks's *Out of the Tarpit* [MM06], and Fred Brooks's *No Silver Bullet* [Bro87].

1.2.1 Defining Complexity

Terminologies describing software complexity are often used casually with vague meaning. Here, we define a few key terms and our intended meaning when used in this thesis. For *simple* and *complex*, we take their definitions from *Simple Made Easy*. For *essential and accidental complexity*, we adapt their definitions from *Out of the Tar Pit*, which themselves are adaptations of *essential and accidental difficulties* from *No Silver Bullet*.

Simple describes the lack of entanglement. A software component is said to be simple if its behavior, role, or idea is not entangled with other components.

Complex is the opposite of simple, describing the entanglement of behaviors, roles, and ideas between software components.

Essential Complexity is complexity inherent in problem to be solved. It is the essence of the problem and cannot be removed.

Accidental Complexity is complexity unrelated to the problem. Moseley and Marks describes it as a mishap, as something non-essential which is present.

The definitions listed here are quite loose. Although there are complexity metrics which attempt to define complexity in an objective manner, we do not find them useful. Complexity metrics are difficult to interpret and are only loosely correlated with software complexity [KST$^+$86]. They are not useful for identify complexity on a daily basis. Despite our broad definitions for complexity, they are a useful starting point for thinking and discussing about complexity in our software.

1.2.2 Simplicity is Key for Development

Moseley, Marks, and Hickey all argue these facts: we must be able to understand our software if we wish to make them robust and correct; our primary method for understanding software is through informal reasoning; and complexity destroys our ability to reason about our software. Complex software is difficult to understand because its components are entangled; as we attempt reason about one component, our mental capacity becomes burdened with additional components that comes entangled with it.

The most fundamental part of developing new software is conceptualizing and understanding its design. In his essay, Brooks writes:

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation."

All other qualities in software, its ease of implementation; correctness; maintainability, derive from the quality of its design. Judging the quality of software design is difficult if we cannot understand how it behaves. If we wish to avoid problems when we develop new software, we need to conceptualize a simple design such that we can reason about it.

1.2.3 Simplicity is Hard

Achieving simplicity requires hard work and recognition, as Moseley and Marks states:

> "significant effort can be required to achieve simplicity…(it) can only be attained if it is recognised, sought and prized."

Unfortunately, there is no technology or technique which will bring simplicity to our software, a sentiment expressed by Brooks. The only way to obtain simplicity is to avoid complexity, which requires an understanding of complexity in our software. As Hickey says in his talk:

> "You have to start developing sensibilities around entanglement…You want to start seeing interconnections between things that could be independent."

When we set out to build software in any particular domain, we need to evaluate each of its complexities. For each we find, we need to judge whether such complexity are essential or accidental. Our reasoning about our software's complexity is vital for building it simply.

1.3 Complexities in Annotation Tools

Annotation tools bring together an interface for text documents and annotations. In this section, we examine various aspects of annotation tools and their associated complexities.

1.3.1 Documents

The purpose of annotation tools is to generate annotations anchored to locations within text documents. Annotation tools apply processes to make documents annotatable and may introduce complexities when doing so.

Tokenization

Many annotation tools preprocess text documents by tokenizing their text content. Complexity is introduced if other behaviors in the tool depends on the tokenization. For instance, some tools restrict the creating annotation information to word tokens. In other tools, documents are visualized as sentence tokens. In these kinds of tools, altering the tokenization algorithm will result in side effects in tool's annotation behavior or document visualization.

Annotatable Regions

Documents may be partitioned into separate annotatable regions. For example, a tool may choose that only paragraph elements are annotatable in HTML documents. Tools need to consider how regions are addressed location-wise with a document, how regions are determined to be annotatable, and how annotations behave between and within regions. These considerations add behaviors and rules to the document, creating complexity. There are many possibilities when dealing with annotatable regions. We can only recommend implementing behaviors and rules which yield the least amount of complexity.

1.3.2 Annotations

Annotations are information attached to text within a text document. They are often used to comment or label their targeted text. There are various choices in their implementation, each with different complexity characteristics.

Inline vs Standoff

Annotations may be represented within the document or outside of it. If the annotations are inline, they are placed within the document in the flow of text. Inlined annotations often appear as markup in document formats such as HTML and XML. If annotations are standoff, they are separate from the document's flow of text. Standoff annotations are often stored in a separate file from the document. The format used by BRAT [bra] is an example of a standoff format.

Inlined annotations are not recommended due to complexity of placing them into the document [PS12]. Because they are placed within the document, inlined annotations are tied to limitations of the document format. For example, creating overlapping annotations is difficult in a HTML document because HTML does not support partially overlapping elements. If given the choice, favor standoff annotations over inlined annotations.

Standoff Positions

Because standoff annotations are separate from the document, they need to specify their positions within the document. For a document with annotatable regions, standoff annotations need to specify the region they address. Within each region, the annotations need specify the span of text which is annotated. These specified positions may have complexity if they rely on behaviors in other parts of the tool. For instance, standoff annotation based on tokens are tied to the tokenization algorithm. They are less flexible than standoff annotations based on character indexes [PS12]. If possible, choose position specifiers that don't depend on behaviors from elsewhere.

Constraints

Annotations may have constraints relative to each other. An example constraint may be to forbid overlapping annotations. Enforcing constraints is an extra source of complexity which ties rules to annotations.

Referential Information

Some annotation tools manage information which refer to annotations. Since they are about annotations, they are tied to annotations they refer to. An example is relational data, where each piece of data may describe something between a set of annotations. Changing the relationship's annotations may influence the relationship itself. For instance, if one of the relationship's annotation is deleted, the relationship may become invalid. The tool may handle this by either deleting the invalid relationship or by not allowing annotations with relationships to be deleted.

1.3.3 Implementation

An annotation tool's implementation is a source of complexity. We list a few implementation choices we have seen which make annotation tools complex.

Formats and Models

Documents and annotations are represented in a wide variety of formats such as plain text, HTML, XML, or PDF. Annotation tools parsed documents and annotations encoded in these formats and covert them into in-memory data structures representing them.

The annotation tool's data format and model are mapped to and from each other, forming a source of complexity. The complexity may be widespread if many formats are supported. If the model changes, all supported formats may have to change as well.

To reduce this complexity, choose formats which require less mapping between the format and the data model. Ideally, choose a format which is a literal syntax of the model's data structures. For example, if the annotation tool is written in JavaScript and its model data structures are JavaScript objects, use JSON as the serialization format. Since JSON is a literal syntax for JavaScript objects, complexity between the format and model is greatly reduced because there is no special effort required for mapping one to the other.

Part of a System

Annotation tools are often pieces in a larger system, where they integrate with databases, servers, and other tools. Common examples are web-based annotation tools, where annotation tools are served from their servers. These integrated tools are difficult to reuse in other systems due to their assumptions on external services. For instance, it is difficult to take an annotation tool which assumes a client-server system and place it on a cloud-based crowdsourcing platform where no explicit server exist.

Visualization Styles

All annotation tools have distinct visualization styles which influence the behaviors of their documents and annotations. For example, some tools visualize a document as rows of sentences. In these tools, the document is tokenized by sentences and annotations have special behavior if they cross between them.

Layers

Annotation tools may visualize different sets of annotations using “layers” on their documents. Layers are useful for comparing different sets of annotations. For example, annotation tools may use layers to compare a set of fuzzy annotations to a set of gold standard annotations.

Layers have complexity in their visualization. Layers in annotation tools are analogous to layers in image manipulations programs. They are implemented as "stacks" of sheets on top of documents, where each sheet contains annotation visualization elements positioned above the annotated text. Because annotations may overlap between layers, the ordering of layers is important. Annotations on upper layers obscure or alter the visualization of annotations in lower layers.

1.4 Conclusion

There are situations where the right tools do not exist and building new tools becomes necessary. Building new annotation tools is a challenge in conceptualizing their designs, requiring an understanding of their complexity. We have outlined complexities in annotation tools to help guide development for new annotation tools. We recommend building new annotation tools with simplicity in mind and avoid unnecessary complexity.

Chapter 2

Notate: Simple Functions for Building Annotation Tools

2.1 Introduction

The web browser is a powerful and versatile platform for implementing annotation tools. Client-server based annotation tools such as BRAT [SPT+12] and WebAnno [YGdCB13] have demonstrated the web browser as an easy to use platform for scaling annotation within a small group of expert users. Crowdsourcing studies [GNWS14, YYSXH10, ZLD+13] have demonstrated further scalability by building annotation tools into crowdsourced microtasks distributed through browser based crowdsourcing platforms. It is evident to us that new annotation tools will be built for the browser and benefit from the browser's interface and scalability.

Most annotation tools implement the following features: they visualize annotations by inlining them into text (often to highlight annotated text), and they create annotations from user selected text. These features allow annotators to intuitively annotate text documents and are found in nearly every interactive annotation tool. If we wish to develop new annotation tools, we will need to implement these features.

Currently, developing new annotation tools is inefficient because there is no reusable implementation for the previously mentioned features. Existing implementations in current tools are not reusable because they are complex; they are entangled with other behaviors in their tool that we are not interested in. Their complexity makes it difficult to isolate and reuse in new tools.

The solution is to develop a reusable library which provides a simple implementation for inlining annotations into text. Unfortunately, inlined annotations is a major source of complexity which entangles the annotations with its text document. In Chapter 1, we recommended that tools should avoid this complexity by using standoff annotations. However, because inlined annotations is such a powerful and intuitive way to visualize annotations, it is an essential feature (and thereby essential complexity) for annotation tools. We still recommend storing annotations in a standoff style, but for visualizing annotations, inlined annotations is unavoidable. If we want to develop a simple library for inlining annotations, we need to handle this complexity in the simplest manner.

In addition, most annotation tools implement the same, uninspired inlined visualization for annotations and we believe there is an opportunity for creating new tools with richer visualizations and interfaces. Most tools highlight annotated text; some tools support labeling annotated text; few tools implement visualizations beyond highlighting and labeling (so far, we have only seen relationships between annotations). We believe the complexity from inlined annotations is so great such that is has prevented tool developers from innovating visualizations. If we can build a simple library for inlining annotations, not only do we make development more efficient, but we also enable the development of more innovative tools. Because it is unentangled with any specific behavior, a simple library should have the power to inline arbitrary structures into text. Developers using this library can inline what is needed to support rich inlined visualizations.

In this chapter, we present our ClojureScript library, *Notate*, which is our solution at providing a simple and reusable implementation for inlining annotation data into text. We designed our library to be simple by leveraging functional programming and abstractions on text data types which it operates on. In the following sections, we discuss its design principles and usage for flexibly building new annotation tools.

2.2 Functional Programming for Browser Applications

Browser based annotation tools are built as single page applications (SPA) in order to provide fluid interfaces for their users. SPAs achieve this by managing application state and render state on the client's browser. Unfortunately, stateful applications are difficult to develop because state adds complexity, making them difficult to understand [MM06]. In order to make development simple, we need to limit state as much as possible.

For our work in this chapter, we limit state through the use of functional programming. This style of programming emphasizes building applications mostly out of pure functions, where functions which return the same value given the same input. Applications built out of mostly pure functions have very little state, with state only existing at the fringes (e.g. database request, rendering operations, etc...) of the applications. When the bulk of the application is comprised of pure function, it easy to reason about and easy to test because pure functions are easy to reason about and easy to test [MM06, Hic11]. If we can build SPAs in a functional style, we can avoid unnecessary state and complexity in our applications.

2.2.1 React

In order to build SPAs in a functional style, we have chosen to use the React JavaScript library [rea]. React is a user interface library which allows developers to build components of their web application in a functional style. Many developers describe React as the *View* in the *Model View Controller* pattern. Each React component must define a render function which receives property values as arguments and returns Virtual DOM (VDOM) as output. The VDOM is React's representation of the browser's Document Object Model (DOM); it allows React to efficiently update the webpage. When a React component is instantiated with property values, its resulting VDOM can be mounted onto the webpage. When a VDOM is mounted, React performs the required DOM operations to render the component on the browser. A React component can be updated by instantiating it with different property values, resulting in a different VDOM. When the new VDOM is mounted, React compares the new VDOM with the previously mounted VDOM and performs the minimal DOM changes needed to update the component. React's diffing of VDOMs allows SPAs to be written in a functional style while maintaining rendering efficiency. By handling DOM updates, React relieves developers of managing state complexity in the DOM, making SPA development simple.

2.2.2 ClojureScript

In the past, SPAs were only written in JavaScript because it was the only supported programming language in web browsers. However, recent advancements in JavaScript engines have made JavaScript extremely fast and have allowed JavaScript to be a compilation target for other languages. For our work in this chapter, we have decided to use ClojureScript [clob] as our implementation language. ClojureScript is a port of the Clojure [cloa] language to the browser with JavaScript as its compilation target. ClojureScript is dialect of the Lisp programming language with an emphasis on functional programming. It makes functional programming idiomatic by providing a large core functional library, immutable collections, functional polymorphism mechanisms, and many other features. In addition, the ClojureScript community has made developing functional SPA easy by embracing React. There are many high quality ClojureScript frameworks and libraries designed to work with React. We have chosen to build our library using ClojureScript because of its support for functional programming and its community libraries for writing React applications.

2.3 Complexities when Inlining Standoff Annotations

2.3.1 Accidental Complexities

Inlining annotation data into text is a complex process involving many different behaviors. It is easy to introduce accidental complexity in its implementation. Here, we outline accidental complexities we have seen in annotation tools. Our library is designed to avoid these complexities completely.

What is Inlined

Most, if not all, annotation tools are built as monolithic systems. They control all aspects of their interface and implement a specific visualization style envisioned by their developers. Their inlining implementation is designed to support the tool's interface and style. Unfortunately, because the tool's components are not designed for reuse in other tools, the inlining implementation is often tied with the tool's style and is difficult to alter to inline something else.

Coupled and Premature Rendering

Inlining is a distinct and separate process from rendering. It is possible to inline elements into text in memory without have it show up on the webpage. However, many annotation tools mix these two processes and immediately render what is inlined. Premature rendering may be unsafe because it may unknowingly trample on view states on the webpage. This complexity is not a problem for many tools because they have complete control of their webpage. However, they are hard to reuse in other tools due to their assumptions of control. In addition, prematurely rendering annotations may be less efficient due to a lack of batching. For example, if a tool creates five annotations at once, it is more efficient to render all annotations in one update than to render five separate times in a row.

Specific to Text Type

We often see tools which are only capable of annotating plain text and offer no support for annotating formatted text. Their inlining implementation operates on concrete types such as strings or DOM Text nodes. They are unable to inline other types of text such as formatted text (such as bolded or italicize text) or annotated text (from another tool or a previous step). These tools lack abstractions for text making them difficult to change to annotate on other types of text.

Irrelevant Parent Element

Browser annotation tools contain their annotated text within a parent DOM element. Often, the tool's inlining implementation expects a DOM element to inline into. However, the DOM element is irrelevent to the concept of inlining into text. The inlining implementation often just operates on the element's children text nodes and the DOM element just happens to contain the text. Their irrelevance can be spotted by examining the tool's standoff format. If the format does not contain information about the element within which the annotations are contained, then the DOM element is irrelevant to the tool's inlining process. The process of inlining annotation data into text can be separated from the process of placing the inlined results into a parent element.

2.3.2 Essential Complexities

We have chosen to show inlined annotations as a feature for our browser based tool. Its implementation creates unavoidable complexity by combining the browser, text, and annotations. Sometimes there are choices between different implementations, each with different complexity. In this section, we list essential complexities related to our library and how it handles them.

Wrap with Element vs Overlay

Annotation tools can use one of two styles to show inlined annotations on the browser: either they wrap annotated text with an inlined element, or they use absolute positioned elements on an overlay layer above the text. Each approach differs in complexity.

In order to wrap text with an element, the tool needs to identity the annotated text, split it from the rest of the text, and create an element to wrap the text. Wrapping text is complex for a few reasons. First, it is limited to the hierarchical structure of HTML, which makes it awkward to implement partially overlapping annotations. Second, it changes text by splitting it and inserting elements into it. This mixes the original text with annotations and creates complexity for the tool. The annotation tool now has to manage text consisting of heterogeneous nodes; some of the nodes are the original text and some are annotation text.

In addition, heterogeneous text nodes cause problems for creating annotations because they complicate standoff position calculations. The user's selection may cross the boundaries between text nodes and the selected nodes may be different. The tool must somehow normalize the user's selection to a standoff position while taking all possible selectable states into account.

Some annotation tools implement an overlay instead of wrapping annotated text with elements. In this style, the tool visualizes an annotated document as two layers, where the bottom text layer contains the annotatable text and the top annotation layer contains absolutely positioned elements for annotation visualization. The advantage of this style is that the tool does not have to change the text in the bottom layer. However, overlays have complexity related to positioning and rendering. Problems occur when the positions of text changes unexpectedly and the positions of visualizations are not adjusted accordingly. Text positions may change if CSS styling changes, user zoom level changes, or element layout changes. Annotation tools implementing overlays often find themselves enforcing all aspects of the webpage in an attempt to freeze text positions.

For Notate, we have chosen to support annotating text by wrapping it with elements over overlaying it with position elements. In our experience, we have found that overlays involve too much complexity from the rendering and positioning of text. In contrast, the complexity of wrapped elements can be mitigated with proper abstractions for text. We explain these abstractions in Section 2.4.

Partially Overlapping Annotations

Partially overlapping annotations are awkward to implement by wrapping text with elements because HTML is hierarchical. The tool either needs to disallow partially overlapping annotations or have methods for dealing with them. Tools which allow partial overlaps often handle them by splitting the wrapping elements of overlapping annotations into pieces. One strategy is to have annotations which are on “top” split annotations which are “below” it and have the top piece wrap the overlapping bottom piece. Another strategy is to merge the attributes of overlapping pieces together.

Notate allows partially overlapping annotations by using abstractions which treat wrapped elements as ordinary, annotatable text. We illustrate these abstractions and how they handle partial overlaps in the Section 2.4.

Visualization Doubles Annotation State

Annotation tools manage a stateful collection of standoff annotations where annotators can add, edit, and delete annotations from. When a tool needs to visualize its annotations, it combines annotations with the text document and renders the result to the webpage using the browser’s DOM API. Because the DOM is a stateful API, the rendered result is a second source of state for annotation tools. The tool must manage two sources of annotation state: its standoff collection and the rendered visualization; and it must keep both states synchronized. Otherwise, the visualization may not match with the corresponding state in the annotation collection.

Notate’s strategy for handling visualization state is to let another library manage it, namely React. In addition, Notate provides pure functions where the process of inlining annotations is stateless. We can avoid doubling annotation state by combining pure functions with React. Notate’s functions are meant to be used within React components, computing the inlined visualization as VDOM elements from the value of the standoff annotation collection. Through this technique, our visualization state becomes derived from our standoff annotation collection state and is automatically managed by React.

2.4 Notate's Design

2.4.1 Library of Primitive Functions

Notate provides pure functions for building annotation tools with. By providing pure functions, it avoids complexity which comes with stateful programming. In addition, it is also designed to be as minimal as possible and so it provides only two primitive functions for annotation tools. One function inlines annotation data into text for an individual DOM element. Another function calculates standoff positions from the user's selection. Developers using Notate must combine these functions with other technologies because these two functions do not build a complete annotation tool by themselves. For instance, since Notate does not handle DOM rendering, we expect developers to combine Notate's functions with third-party rendering libraries such as React.

2.4.2 Standoff Annotations and Selection

Notate uses standoff annotations by character indexes, an annotation style common in many annotation tools. In this style, annotated text are specified by an integer interval containing a start and end index. Notate expects an interval to be an integer pair wrapped in a sequence. The first integer in the sequence is the starting index, inclusive. The second integer is the ending index, exclusive. For our examples in this chapter, we use ClojureScript vectors literals (square brackets) wrapping two integers to represent intervals. For example, the vector [0 4] annotates character starting from character 0 and up to, but not including, character 4.

Most annotation tools specify custom formats for encoding standoff annotations. Despite their intricacy, standoff annotation formats can be broken into two distinct parts: character index interval and domain information. Notate does not deal with annotation formats or domain specific information. Instead, Notate provides a function for creating character index intervals from user selected text. It is left to the developer to use this function to build a complete annotation with domain information.

2.4.3 Inlining Annotation Data into Text

Notate provides a function, *inline*, which inlines standoff annotation data into text by splitting it. Consider the HTML element in Listing 2.1. If we wish to wrap *dddd* with an element, we specify the interval [12 16] to *inline*. It then splits *dddd* from the rest and wraps it with a <*mark*> element (Listing 2.2).

Listing 2.1: HTML element example

```
<div>aaaabbbbccccddddeeee</div>
```

Listing 2.2: Element annotated at interval [12 16]

```
<div>aaaabbbbcccc<mark>dddd</mark>eeee</div>
```

inline works with formatted text as well. Formatting introduces a few challenges for inlining. In Listing 2.3, *bbbb* is emphasized, causing the div element to contain the following three child nodes: *aaaa*, <*em*>*bbbb*</*em*>, and *ccccddddeeee*.

Listing 2.3: HTML element with formatting

```
<div>aaaa<em>bbbb</em>ccccddddeeee</div>
```

inline must be able to annotate a heterogeneous sequence of text-like nodes. In order to wrap *dddd* at interval [12 16], it needs to determine its location in the third node. This is accomplished by obtaining the logical character lengths of the nodes. In our example, the *aaaa* node has a length of 4, the *<em>bbbb</em>* node has a length of 4, and the *ccccddddeeee* node has a length of 12. *inline* uses the nodes' lengths to assign cumulative intervals. The nodes *aaaa*, *<em>bbbb</em>*, and *ccccddddeeee* are assigned the intervals [0 4], [4 8], and [8 20] respectively. The annotation interval and the node intervals are used to determine which nodes are splitted and wrapped.

Annotations may partially overlap with formatting. In this scenario, formatted text must be splittable, like all other text nodes. In Listing 2.4, an annotation of interval [6 16] overlaps with an emphasis node, which has an interval [4 8]. The result is that the emphasis node is split in half, with the second half enclosed by the annotation's wrapping element.

Listing 2.4: HTML element with formatting

```
<div>aaaa<em>bb</em><mark><em>bb</em>ccccdddd</mark>eeee</div>
```

Our formatting examples are actually a specific scenario of a more generic scenario. As long as all nodes have character lengths and are splittable, they can be inlined into. In addition, the parent node (the *div* element in our examples) is irrelevant to annotation, only its children nodes matter. Therefore, *inline* uses this abstraction: any sequence of lengthed and splittable nodes can be inlined into.

inline's abstraction allows it to easily implement a difficult feature in annotation tools: layering. As long as the inlined result is a sequence of lengthed and splittable nodes, it can be inlined into again. By using this technique, an annotation tool can logically organize different groups of annotations together and inline them in stages by chaining *inline* calls one after another.

So far in our examples, we have annotated by enclosing nodes within a <*mark*> element. However, annotation tools may need to inline arbitrary data into text in order to support more advanced features. For example, a tool may wish to have annotated text adorned with labels. In Listing 2.5, the label text “foobar” is inlined to the left of the annotated text. Using CSS, the label text can be styled on top, as shown in Figure 2.1. Notate allows arbitrary inlining by making *inline* a higher-order function, where one of its argument is a function which determines what is inlined. Developers can control the return value of this function and inline what is needed for their annotation tool.

Listing 2.5: Annotated text with label text to its left.

```
<div>aaaabbbbcccc<span class=‘‘annotation’’><span class=‘‘label’’
    >foobar</span><mark>dddd</mark></span>eeee</div>
```

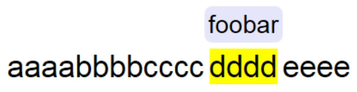


Figure 2.1: Rendering of Listing 2.5 with CSS styling.

2.4.4 Creating Intervals from User Selection

Notate provides a function, *range-interval*, for creating character intervals from user selected text. This function uses the browser’s Range object interface to determine what DOM elements are selected. The Range object has two properties, startContainer and endContainer, which specifies which DOM nodes start and end the user’s selection. The Range object also contains two additional properties, startOffset and endOffset, which specifies the offset of startContainer and endContainer respectively. *range-interval* uses these four properties to calculate the equivalent selected interval. The indexes for the interval are calculated by adding the offset to the sum of character lengths for each of the container's previous sibling nodes to parent node.

Consider the example in Listing 2.6 where the user has selected *dddd*. In this example, both of the Range's startContainer and endContainer are the node *ccccddddeeee*, and both startOffset and endOffset are 4 and 8 respectively. To calculate the start index, the range-interval function sums the length of the startContainer's previous siblings with the startOffset. The startContainer's previous siblings are *aaaa* and *<em>bbbb</em>*, each with a length of 4. The resulting start index is 12. The same process is repeated for the end index, resulting in an end index of 16. The interval for this example selection is [12 16].

Listing 2.6: The bar character, |, shows where the user has selected and is not part of the text. In this example, *dddd* is selected.

```
<div>aaaa<em>bbbb</em>cccc|dddd|eeee</div>
```

Sometimes, there are insertions of extra "meta" text that are not logically part of the original, annotated text. These meta text throws off interval calculations by adding extra lengths to the sum. For example, developers may choose to insert extra label text as explained previously (Listing 2.5). Invisible nodes with character text, such as comment nodes, also throw off interval calculations. In order to correct interval calculations, Notate provides a mechanism for controlling a node's length. Developers can use this mechanism and set meta nodes to have a length of zero. By using this technique, *range-interval* is able to calculate the correct interval even in the presence of extra meta nodes.

2.5 Protocol and Function Definitions

2.5.1 Protocols

In ClojureScript, a protocol [proa] is a set of extensible functions used for polymorphism, similar to interfaces in other languages. Protocol functions are dispatched based on the type of their first argument. A type may extend to a protocol by providing implementations for the protocol's functions. Other functions may use protocols as an abstraction by calling the protocol's functions with an extending type, without knowledge of function's implementation or type.

Notate defines two protocols, *Lengthed* and *Splittable*, which are used as abstractions for lengthed and splittable nodes mentioned in Section 2.4. These two protocols are used by *inline* and *range-interval*. Notate provides default implementations for a few common types.

Lengthed

The protocol *Lengthed* (Lisiting 2.7) defines one protocol function, *length*, which is used to obtain the character length of a text-like node.

Listing 2.7: Lengthed protocol

```
(defprotocol Lengthed
  (length [this] ``Character length of this.''))
```

Types extending *length* are expected to return their character length as an integer. *Lengthed* is used by *inline* and *range-interval* functions. For *inline*, Notate provides default implementation of *length* for strings and vectors, which are used to represent React's VDOM Text and VDOM Element nodes respectively. For *range-interval*, Notate provides default implementations of *length* for DOM Text, DOM Elements, and DOM Comments. Listing

2.8 shows examples of *length* dispatched to Notate's default implementation for strings and vectors.

Listing 2.8: Example results of *length* for strings and vectors.

```
=> (length ``foobar'')
;; returns 6

=> (length ``foobarbaz'')
;; returns 9

=> (length [:span ``foo''])
;; returns 3

=> (length [:span ``foo'' [:mark ``bar'']])
;; returns 6
```

Splittable

The protocol *Splittable* (Lisiting 2.9) defines one protocol function, *split*, which is used to split text-like nodes at the given indexes.

Listing 2.9: Splittable protocol

```
(defprotocol Splittable
  (split [this indexes] ``Split this at indexes.''))
```

Types extending *split* are expected to split themselves into a sequence at the given *indexes* argument. *Splittable* is only used by *inline*. Default implementations for *split* are provided for strings and vectors. Listing 2.10 shows examples of *split* dispatched to Notate's default implementation for strings and vectors.

Listing 2.10: Example results of *split* for strings and vectors.

```
=> (split ``foobar'' [3])
;; returns (``foo'' ``bar'')

=> (split ``foobarbaz'' [3 6])
;; returns (``foo'' ``bar'' ``baz'')

=> (split [:span ``foobar''] [3])
;; returns ([:span ``foo''] [:span ``bar''])

=> (split [:span ``foo'' [:mark ``barbaz'']] [3 6])
;; returns ([:span ``foo''] [:span [:mark ``bar'']]
            [:span [:mark ``baz'']])
```

2.5.2 Functions

inline

Notate provides the function *inline* for inlining standoff annotation data into sequence of text-like nodes (called *content*). *inline* accepts four arguments shown in its definition in Listing 2.11. The first argument, *f*, is a function which defines what is inlined into *content*. *f* is a function of two arguments, where the first is the annotation data and the second is the enclosing, splitted content. The return value of *f* should be a sequence which is inlined into the text content. The second argument, *offset*, is an integer offset for *content*. The third argument, *ianns*, is a sequence of interval-annotation data pairs representing standoff annotations. For each pair in *ianns*, function *f* is called with the annotation data and the enclosing text content specified by the interval. The last argument, *content*, is a sequence of *Lengthed* and *Splittable* nodes being annotated.

Listing 2.11: *inline*'s function signature. See Appendix A for implemented algorithm.

```
(defn inline
  ``Inlines data into content given intervaled data.''
  [f offset ianns content]
  ...)
```

range-interval

Notate provides the function *range-interval* for calculating a character interval from the user's selection. *range-interval* accepts two arguments, *node* and *range-obj*. *node* is the DOM node which the interval is calculated relative to. *range-obj* is the browser's Range object which represents the user's selection. *range-obj* must be contained within *node*.

Listing 2.12: *range-interval*'s function signature. See Appendix A for implemented algorithm.

```
(defn range-interval
  ``Normalize range into an interval relative to node.''
  [node range-obj]
  ...)
```

2.6 Function Usage

2.6.1 inline

We expect developers to embed *inline* calls within their web application component framework/library, be it React or anything else. Although *inline* is flexible enough to work with any component system, Notate provides convenient defaults for working with React. Notate represents markup by using vectors with Hiccup syntax [Ree]. In Hiccup syntax, the first item in the vector is a keyword which designates the element's name. The second item may be an optional map representing the element's attributes. The remaining items are the element's children. Hiccup syntax is convenient for building React applications in ClojureScript because it can be compiled into React VDOM nodes through the Sablono library [r0m]. Here, we show how to use *inline* to implement the results similar to the ones shown in Section 2.4.3. However, instead of wrapping text with markup, we wrap them using Hiccup vectors.

In Listing 2.13, we enclose the content at interval [12 16] with a vector representing the <*mark*> element, analogous to the example from Listing 2.2.

Listing 2.13: Annotating with Hiccup vector syntax.

```
=> (inline (fn [ann cnt]
               [(into [:mark] cnt)])
             0
             [[[12 16] nil]]
             [``aaaabbbbccccddddeeee''])
;; returns (``aaaabbbbcccc'' [:mark ``dddd''] ``eeee'')
```

Notate implements *Lengthed* and *Splittable* for vectors with *Hiccup* semantics. This allows *inline* to annotated formatted text as shown in Listing 2.14.

Listing 2.14: Annotating formatted text with partial overlap.

```
=> (inline (fn [ann cnt]
             [(into [:mark] cnt)])
           0
           [[[6 16] nil]]
           [''aaaa'' [:em ''bbbb''] ''ccccddddeeee''])
;; returns (''aaaa''
            [:em ''bb'']
            [:mark [:em ''bb''] ''ccccdddd'']
            ''eeee'')
```

Developers can inline arbitrary structure into *content* by supplying the appropriate function *f*. Listing 2.15 shows how to inline a label structure into the text content.

Listing 2.15: Annotating a label structure into content.

```
=> (inline (fn [ann cnt]
             [[:span {:class ''annotation''}
                [:span {:class ''label''} ann]
                (into [:mark] cnt)]])
           0
           [[[12 16] ''foobar'']]
           [''aaaabbbbccccddddeeee''])
;; returns (''aaaabbbbcccc''
            [:span {:class ''annotation''}
              [:span {:class ''label''} ''foobar'']
              [:mark ''dddd'']]
            ''eeee'')
```

inline function calls can be chained together as long as the annotated result after each step is a sequence of *Lengthed* and *Splittable*. In Listing 2.16, two inline calls are chained together using ClojureScript's thread-last macro (->>), where the result of each function call is fed as the last argument into the next expression. Because the example inlines using vectors, each inlined result is a sequence of *Lengthed* and *Splittable*, allowing it to be inlined again.

Listing 2.16: Chaining two inlines together using thread-last macro.

```
=> (->> [''aaaabbbbccccddddeeee'']
     (inline (fn [ann cnt] [(into [:em] cnt)]) 0 [[[4 8] nil]])      ; #1
     (inline (fn [ann cnt] [(into [:mark] cnt)]) 0 [[[6 16] nil]])) ; #2

;; After #1, inline returns (''aaaa''
                            [:em ''bbbb'']
                            ''ccccddddeeee'')
;; After #2, inline returns (''aaaa''
                            [:em ''bb'']
                            [:mark [:em ''bb''] ''ccccdddd'']
                            ''eeee'')
```

2.6.2 range-interval

range-interval should be used when the annotation tool needs to create annotations from the user's selection. From our experience, we have used *range-interval* within callback functions attached to event listeners. The callback calls *range-interval* to compute the user's selected interval and adds it to a data structure required for representing an annotation (specific to the tool, but often a map), along with any relevant domain or event specific information.

Notate provides default *length* implementations for DOM Text, DOM Elements, and DOM Comments, shown in Listing 2.17. A DOM Text's length is the character length of themselves. A DOM Element's length is the total length of their child nodes. A DOM Comments have a length of zero. These implementions work well if the text does not contain "meta" text nodes as mentioned in Section 2.4.4.

Listing 2.17: Default implementation of length for DOM nodes.

```
(extend-protocol Lengthed
  js/Text
  (length [this] (count (.-wholeText this)))
  js/Element
  (length [this] (total-length (array-seq (.-childNodes this))))
  js/Comment
  (length [this] 0))
```

If there is meta text, the implementations under *Lengthed* needs to be adjusted in order for *range-interval* to correctly compute intervals. Developers can adjust this by re-extending the protocol again. In our experience, we have found that we only need to adjust the implementation for DOM Text because the default length of DOM Element is derived from the sum of its children elements. When DOM Text is adjusted, DOM Element is automatically adjusted as well. In Listing 2.18, we re-extend *length* for DOM Text such that if its parent element contains the class "label", it has a length of zero. In this example, if the user selects text under label elements, elements containing label elements, or text preceded by label elements, the length of the label elements will not count towards the interval calculation.

Listing 2.18: Ignore DOM text nodes whose parent element is a label.

```
(extend-protocol Lengthed
  js/Text
  (length [this]
    (if (.. this -parentNode -classList (contains ''label''))
      0
      (count (.-wholeText this)))))
```

2.7 Demonstration Tools

In this section, we demonstrate a variety of annotation tools built with Notate. We start with basic highlighting tools, followed by tools with advancing capability. All demonstration tools are built using *Rum* [Prob], a ClojureScript React library for quickly building SPA applications.

2.7.1 Basic

In this tool, we show Notate's most basic use case: creating and highlighting annotated text from the user's selection. Annotated text are wrapped in a HTML mark element giving it a yellow background color (Figure 2.2). The user creates annotations by selecting text with his mouse. Annotations are deleted by clicking on the highlights. The basic tool can also highlight across formatted text (Figure 2.3).

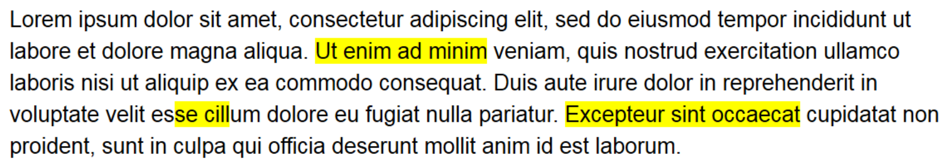

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 2.2: The basic annotation tool highlights annotated text.

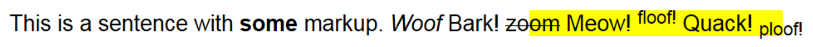
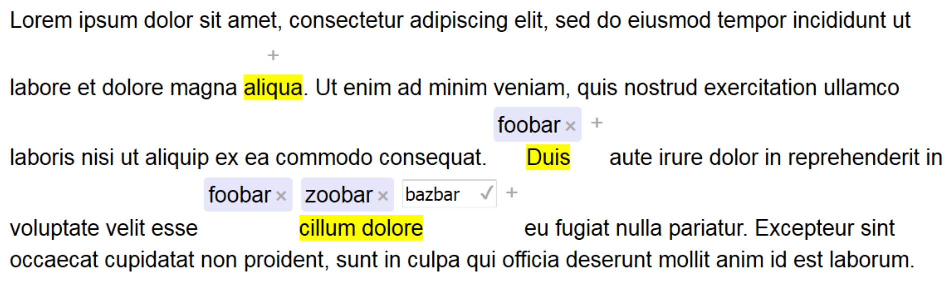

This is a sentence with **some** markup. *Woof* Bark! ~~zoom~~ Meow! $^{\text{floof!}}$ Quack! $_{\text{ploof!}}$

Figure 2.3: The basic annotation tool can annotate formatted text as well.

2.7.2 Labeling

Here, we demonstrate Notate's ability to implement an interactive labeling tool. In this tool, annotated text are inlined with multiple interactive labels and buttons. The user annotates like in the basic demo and can add labels to the annotations. Labels are added by clicking on the "plus" button above the annotated text. When a label is added, the user's input is focused onto an input box which appears above the annotated text. Label text can be typed into the box and confirmed with the "check" button. The user can delete the label with "cross" button.


Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut

+

labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco

foobar × +

laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in

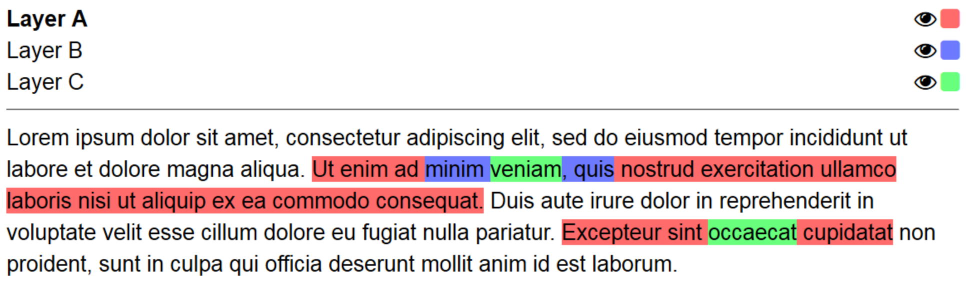
foobar × zoobar × bazbar ✓ +

voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 2.4: The interactive labeling tool allows users to label text without popup menus.

2.7.3 Layering

This tool demonstrates Notate's ability to build annotation tools with layers. The tool contains three layers, distinguished by their colors. Each subsequent layer is "on top" of the previous layer, hiding the color of the layer beneath it. In the example shown in Figure 2.5, Layer A sits on the bottom, Layer B is on top of A, and Layer C is on top of B. We attached a control menu on top of the annotatable text. The user can use this menu to change the color and visibility of each layer.

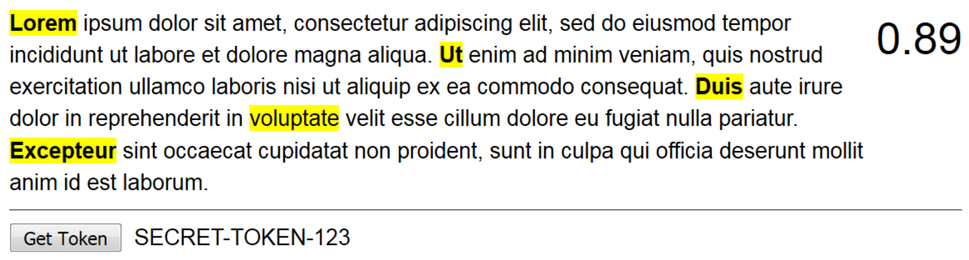


Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 2.5: The layering annotation tool has three layers of annotations. The user can change the current annotating layer, layer visibility, and layer colors with the control menu on top.

2.7.4 Scoring

We can imagine annotation tools being used to train or filter candidate annotators for annotation projects. The scoring demonstration tool (Figure 2.6) hides a secret token from the candidate annotator. The candidate is scored on his performance relative to a known gold standard. If the candidate's score is high enough, he can retrieve the token by clicking the "Get Token" button. In this example, the tool calculates the candidate's $F_1 score$ relative to annotations matching the bolded words.


Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. **Ut** enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. **Duis** aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. **Excepteur** sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

0.89

SECRET-TOKEN-123

Figure 2.6: The scoring annotation tool can be used to score candidates for annotation projects. The candidate can retrieve a secret token if his score is high enough.

2.7.5 Suggesting

Manual annotation tools can work in combination with machine learning annotation algorithms. These algorithms often compute the confidence of their annotations ranging from weak to strong. If the algorithm annotates unreliably, its results need to be validated by human annotators. A manual annotation tool can be used to present automatic annotations as suggestions for human annotators to validate.

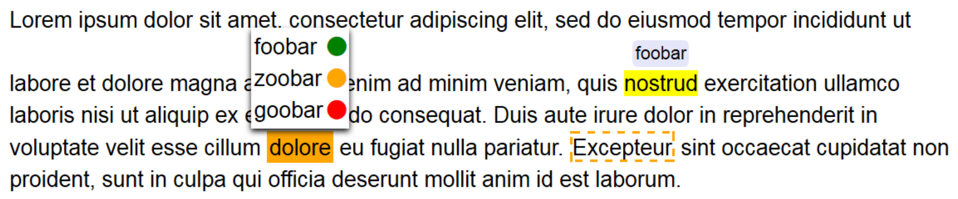

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna a enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex e do consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 2.7: The suggesting annotation tool presents suggestions to the user. The user can click on the bordered dotted text to show the suggested labels. The user can then click on a suggested label to annotated the dotted text.

In our suggestion example (Figure 2.7), we imagine an annotation tool which presents documents that have been pre-annotated with suggestions from an automatic algorithm. These suggestions are rated by their confidence scores, shown by the colored circles. The suggestions are displayed on their own layer and are indicated by their orange dotted borders. Annotators can open the suggestion menu by clicking on the dotted bordered text. Clicking on items in the suggestion menu will annotate the text with the suggested label. A separate layer is used to show annotated text with yellow highlights and labels.

Our suggestion example demonstrates Notate's flexibility and simplicity by combining arbitrary inlining, chained inlined calls for layering, and meta-text ignoring. In the suggestion layer, text is inlined with a hidden, interactive suggestion menu. In the annotated layer, text is inlined with labels. Despite the intricacies of this demo, we have found that it easy to develop due to Notate's simplicity.

2.8 Conclusion

It is likely that new annotation tools will be built on the browser. In addition, new tools will most likely feature inlined annotations as it is an intuitive way to visualize annotations. However, developing new browser annotation tools is inefficient because there are no reusable implementations for inlining annotation data into text. Existing implementations are complex and hard to reuse.

We overviewed sources of complexity when inlining annotations on browser based annotation tools. We have use this to guide our development of our library, Notate, whose goal is to make the development of annotation tools as easy as possible by providing a reusable implementation for inlining. Notate is designed to be simple and its simplicity gives it great flexibility. We shown how to use Notate and built demonstration tools proving its flexibility.

Notate will inspire developers to create new and innovative annotation tools. Further, Notate is our evidence that complexity management and abstractions is important for building new annotation tools. We encourage annotation tool developers to be mindful of complexity in their tools and we hope Notate's design serves as an inspiration on how to handle complexity for future tools.

Appendix A

Function Algorithms

A.1 inline

A.1.1 Algorithm

Inputs

- f: Function of two arguments: *ann* and *cnt*. Must return a sequence of *Lengthed* nodes.
- *offset*: An integer offset of *content*.
- *ianns*: A sequence of interval-annotation pairs.
- *content*: A sequence of *Lengthed* and *Splittable* nodes.

Output

- A sequence of nodes.

Protocols

- *Lengthed*
  - *length*: A function of one argument: *this*. Returns an integer.
- *Splittable*
  - *split*: A function of two arguments: *this* and *indexes*. Returns a sequence of nodes.

Steps

1. Let $content_{smap}$ be a sorted map. For each node n_i in *content*, insert a key-value pair into $content_{smap}$ where the key is the interval $[start_i\ end_i]$ and the value is n_i. If $i = 0$, then $start_i = \mathit{offset}$, otherwise $start_i = end_{i-1}$. $end_i = start_i + (length\ n_i)$.

2. Reduce over the sequential values in *ianns* with reduction function *rf* and an initial value of $content_{smap}$. Let $[itvl_{ann}\ ann]$ be the interval-annotation pair in *ianns* and $content_{smap\_rf}$ be the reduction value *rf* is currently reducing over. Reduction function *rf* has the steps in the following subsection:

   (a) Query $content_{smap\_rf}$ to find key-value entries $entries_{overlaps}$ which have overlapping keys with $itvl_{ann}$.

   (b) Dissociate keys in $content_{smap\_rf}$ with the keys in $entries_{overlaps}$.

   (c) Let $[start_{ann}\ end_{ann}]$ be the indexes in $itvl_{ann}$. Map over the key-value entries in $entries_{overlaps}$. Let *ent* be the current key-value entry being mapped over, with *ent*'s key as $[start_n\ end_n]$ and value as n. Let *indexes* be a sequence containing the integers $start_{ann}$ and/or end_{ann} if they are exclusively between $start_n$ and

end_n. If the size *indexes* is zero, return *ent* wrapped in a sequence. Otherwise, return the result from the steps in the following subsection:

i. Map over *indexes*, subtracting $start_n$ from each index.

ii. Let $indexes_{norm}$ be the result from step 2(c)i. Call the function *split* as shown with the arguments: $(split\ n\ indexes_{norm})$.

iii. Map the splitted nodes from step 2(c)ii. Let m_i be the current node being mapped over. For each node, return a key-value entry with a key of $[start_i\ end_i]$ and value of m_i. If $i = 0$, the $start_i = start_n$, otherwise $start_i = end_{i-1}$. $end_i = start_i + (length\ m_i)$.

(d) Concatenate the sequence of sequences of entries from step 2c into a sequence of entries.

(e) Map over the entries from step 2d, assigning each entry with an identifier called *group*. Let $[start\ end]$ be the interval key of the current entry being mapped over. If $start_{ann} \leq start \leq end \leq end_{ann}$, the *group* has the value of $[start_{ann}\ end_{ann}]$. Otherwise, group as the value of nil.

(f) Partition the entries from step 2e by their *group* identifier.

(g) Map over the sequence of sequences of entries from step 2f, removing the group identifier.

(h) Map over the sequence of sequences of entries from step 2g. For each sequences of entries *soe* being mapped over, assign the sequences of entries its spanning-interval $[start\ end]$. The spanning-interval's *start* index is the start index of the first entry's interval in *soe*. The spanning-interval's *end* index is the end index of the last entry's interval in *soe*.

(i) Map over the sequence of sequences of entries from step 2h. For each sequences of entries *soe* currently being mapped over, if *soe*'s spanning-interval is equal to $itvl_{ann}$, return the result from steps in the following subsection. Otherwise, return *soe* unchanged.

  i. Map over the entries in *soe*. For the entry being mapped over, obtain the entry's value.

  ii. Let *cnt* be the sequence of content nodes from step 2(i)i. With the *ann* from step 2, call the function *f* as shown with the arguments: $(f\ ann\ cnt)$.

  iii. Let $start_{soe}$ be the start index of *soe*'s spanning-interval. Map over the result from step 2(i)ii. For each node n_i currently being mapped over, return a key-value entry with an key of $[start_i\ end_i]$ and value of n_i. If $i = 0$, then $start_i = start_{soe}$, otherwise $start_i = end_{i-1}$. $end_i = start_i + (length\ n_i)$.

(j) Concatenate the sequence of sequences of entries from step 2i into a sequence of entries.

(k) Associate each entry from step 2j into $content_{smap\_rf}$.

3. Let $content_{smap\_reduced}$ be the sorted map returned by the reduce in step 2. Return the sequence of values in $content_{smap\_reduced}$ by mapping each key-value entry for its value.

A.2 range-interval

A.2.1 Algorithm

Inputs

- *node*: A DOM node.
- *range-obj*: A Range object contained within *node*.

Output

- A vector with two integers representing an interval.

Protocols

- *Lengthed*
  - *length*: A function of one argument: *this*. Returns an integer.

Steps

1. To compute the interval's *start* index, perform the steps in the following subsection with *range-obj*'s *startContainer* and *startOffset*:
   (a) Let *A* be a sequence of ancestor nodes from *startContainer* to *node*, inclusive and exclusive respectively. Mapcat over nodes in *A*. For each node *a* in *A* being mapped over, return *a*'s previous siblings as a sequence of nodes.
   (b) Normalize *startOffset* by performing the following. If *startContainer* is a Text or Comment node, then the normalized value is *startOffset* itself, unchanged. Else,

let *children* be the first *startOffset* number of child nodes in *startContainer*; let *c* be a child node in *children*; the normalized value for *startOffset* is:

$$\sum_{i=0}^{startOffset} (length\ c_i)$$

(c) Let *P* be the sequence of previous siblings from step 1a, *offset* be the normalized offset from step 1b, *n* be the size of *P*, and *p* be a node in *P*. The *start* index is the following:

$$min((length\ startContainer),\ offset) + \sum_{i=0}^{n}(length\ p_i)$$

2. Repeat step 1 to compute the *end* index, using *endContainer* and *endOffset* instead.

3. Return [*start end*] as the interval.

Bibliography

[bra] Standoff format - brat rapid annotation tool. http://brat.nlplab.org/standoff.html. Accessed: 2016-11-9.

[Bro87] F Brooks. *No silver bullet*. April, 1987.

[Bur12] Manuel Burghardt. Usability recommendations for annotation tools. In *Proceedings of the Sixth Linguistic Annotation Workshop*, pages 104–112. Association for Computational Linguistics, 2012.

[Bur14] Manuel Burghardt. *Engineering annotation usability-Toward usability patterns for linguistic annotation tools*. PhD thesis, 2014.

[cloa] Clojure. https://clojure.org/. Accessed: 2016-10-17.

[clob] Clojurescript. http://clojurescript.org/. Accessed: 2016-10-17.

[FE16] Mark A Finlayson and Tomaž Erjavec. Overview of annotation creation: Processes & tools. *arXiv preprint arXiv:1602.05753*, 2016.

[GNWS14] Benjamin M Good, Max Nanis, CHUNLEI Wu, and ANDREW I Su. Microtask crowdsourcing for disease mention annotation in pubmed abstracts. In *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, pages 282–293. NIH Public Access, 2014.

[Hic11] Rich Hickey. Simple made easy. *Strange Loop 2011*, 2011.

[KST$^+$86] Joseph P Kearney, Robert L Sedlmeyer, William B Thompson, Michael A Gray, and Michael A Adler. Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050, 1986.

[MM06] Ben Moseley and Peter Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006.

[proa] Clojure - protocols. http://clojure.org/reference/protocols. Accessed: 2016-10-17.

[Prob] Nikita Prokopov. tonsky/rum: Simple, decomplected, isomorphic html ui library for clojure and clojurescript. https://github.com/tonsky/rum. Accessed: 2016-10-17.

[PS12] James Pustejovsky and Amber Stubbs. *Natural language annotation for machine learning*. " O'Reilly Media, Inc.", 2012.

[r0m] r0man. r0man/sablono: Lisp/hiccup style templating for facebook's react in clojurescript. https://github.com/r0man/sablono. Accessed: 2016-10-17.

[rea] A javascript library for building user interfaces | react. https://facebook.github.io/react/. Accessed: 2016-10-17.

[Ree] James Reeves. weavejester/hiccup: Fast library for rendering html in clojure. https://github.com/weavejester/hiccup. Accessed: 2016-10-17.

[SPT+12] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. Brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107. Association for Computational Linguistics, 2012.

[YGdCB13] Seid Muhie Yimam, Iryna Gurevych, Richard Eckart de Castilho, and Chris Biemann. Webanno: A flexible, web-based and visually supported system for distributed annotations. In *ACL (Conference System Demonstrations)*, pages 1–6, 2013.

[YYSXH10] Meliha Yetisgen-Yildiz, Imre Solti, Fei Xia, and Scott Russell Halgrim. Preliminary experience with amazon's mechanical turk for annotating medical named entities. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, pages 180–183. Association for Computational Linguistics, 2010.

[ZLD+13] Haijun Zhai, Todd Lingren, Louise Deleger, Qi Li, Megan Kaiser, Laura Stoutenborough, and Imre Solti. Web 2.0-based crowdsourcing for high-quality gold standard development in clinical natural language processing. *Journal of medical Internet research*, 15(4):e73, 2013.