Semantic Editing in Word Processors

Hristo Staykov

Advisor: Prof. Holly Rushmeier

CPSC 490
Final Report

Abstract

Text selection in word processors is ordinarily done using a pointing device or the keyboard arrow cluster, and can be precisely defined on a character basis. However, such precision is unnecessary if user thinks of the text as a string of words or sentences. It can impose a burden when the user is trying to select a complete sentence, but has to point to the exact start and end of the desired selection. This can be particularly troublesome on mobile devices with limited screen space.

This project explores how tools from the NLP community can aid the user in editing text by guiding their selections. There already exist shortcuts for selecting single words or the entirety of the text, but, to the author's knowledge, there are no tools which allow for quick selection of sentences, embeded sentences, and constituents in natural language.

This project implements a simple text editor that processes text using a constituency parser. Employing the text's hierarchical structure, this editor augments various familiar interfaces for selecting text, and to proposes a few new ones. Moreover, the editor provides visual hints towards the structure of the text in the form of syntax highlighting, which in our testing facilitated skim-reading and guided the user when selecting text.

Chapter 1

Semantic Editing in Word Processors

This is the report for my CPSC 490 Project. To read, simply open <./main.pdf>.

To make:

- `make` will make `main.pdf` from the latest source. `pandoc` and a working TeX installation are required.
- `rm -r aux .deps/*` will clear the cache

Chapter 2

Introduction

Most text editing interface in use today derives from the Gypsy editor developed in the 1970s, where text is selected by dragging the mouse [12].

Modern operating systems offer additional shortcuts to aid with selection, such as double-clicking to select a word and triple-clicking to select a paragraph.

However, for any other type of selection, the user has to rely extensively on a pointing device or the keyboard arrow cluster. Consider the following passage, where the user selection is shown in grey:

> I rode my bike one last time though the maze of small streets and alleys, stopped beside the waffle shop, and flipped the kickstand.

Suppose we want to select the phrase the maze of small streets and alleys. This can be done in one of two ways:

- Using the keyboard, move the cursor to the beginning of the desired selection, and, using the ⇧ + → keys, select to the end.
- Using the mouse, move the cursor to the start of the selection, press the left button, and drag to the end of the selection, and release.

With either method, it is possible to miss or add an extra character, and create selections such as he maze of small streets and alleys, .

2.1 Hierarchial structure within sentences

Natural text has underlying hierarchical structure. In linguistics, semantically coherent groupings are referred to as "constituents" [11]. In the sentence above, the selection streets is contained in multiple constituents:

- "small streets"
- "small streets and alleys"

- "the maze of small streets and alleys"

Determining whether a string of words constitutes a constituent can be done using constituency test. One such test is substitution: constituents can often be replaced by a single word with the same meaning. For example, the last constituent in the list above ("the maze of small streets and alleys") can be replaced with the noun "campus", to form the sentence "I rode my bike one last time through *campus*, stopped beside..."

Observe that the current selection (streets) is already contained within the desired final selection. Moreover, the desired selection happens to form a *constituent* in the syntax tree of the sentence. We speculate that the majority of selection the user makes in a text are in fact constituents: single words in a sentence, a complete sentence, or smaller groupings in-between. We propose several user interactions to exploit that structure, described in sec. 4.

2.2   Previous work

Some operating systems such as macOS have the notion of different granularities when selecting text. Double clicking on a word in a text field selects the word and sets the granularity to *word selection*, and thus subsequent clicks while holding the ⇧ key will expand the selection to include the word under the cursor. The `NSTextView` component, which forms the basis for all text fields in macOS, has three such granularities: selection by characters (default), selection by words (activated with a double click), and selection by paragraphs (activated with a triple click) [1].

Modal code editors like `vim` and `kakoune` support selecting so-called *"text objects"*, such as a line of code, or characters enclosed between delimiters such as *"("* and *")"* [13][7]. More recently, `tree-sitter`[3] has been used alongside such editors to parse the code being edited and provide more intelligent selections.

However, the approach taken in modal text editors does not translate well to editing natural language:

- Natural language is not as structured as source code, and cannot be parsed as easily in real time. Punctuation alone is not enough to delimit sentences.
- The keyboard shortcuts for creating such selections are not discoverable by the average user.

Parsing of natural language is can be realised through the use of *constituency parsers* and *dependency parsers*. Such parsers exist for various languages and can reach parsing speeds up to hundreds of sentences in a second on consumer hardware (See sec. 3.3.1).

Chapter 3

Implementation

3.1 Overview

We implement a simple text editor consisting of a single text field. Certain behaviours of the text field have been overriden to provide interactions for selecting constituents (see sec. 4), and text is highlighted to show its hierarchical structure and to bring focus to the sentence that's being edited.
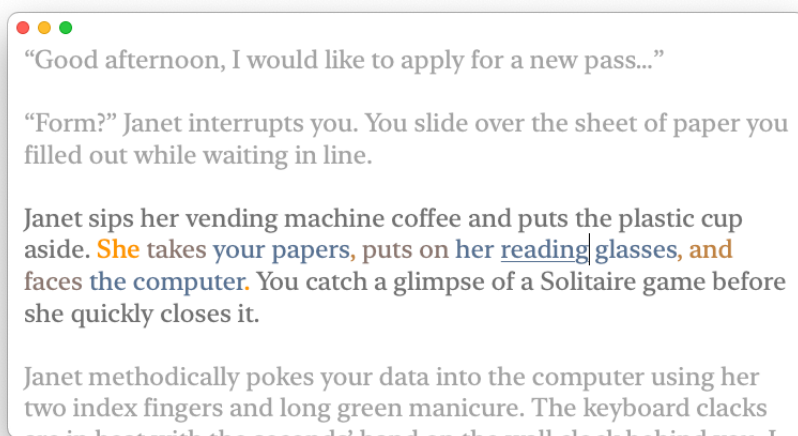

Figure 3.1: Screenshot of our text editor editing prose

The editor is implemented in two layers: a *back end*, which handles parsing, and a *front end*, which presents text and handles user input.

3.2   Front end

We chose to implement the application as an AppKit application for macOS due to the author's familiarity with Swift. The text field is subclass of `NSTextView` that calls the parser whenever the texts has been changed.

3.3   Back end

3.3.1   Choice of parser

We use an existing implementation with a pretrained model for a constituncy parser. We experimented with a few constituency parsers for this application.

3.3.1.1   Stanford CoreNLP

First, we looked into Stanford's CoreNLP package, which is a Java program that can run as a server locally on the user's computer. It encodes results in a protobuffer, and we were able to easily create an API for CoreNLP in Swift (See [6]).

CoreNLP also outputs offsets for teach token in the input text, which makes it easy to parse the parse to the original document.

However, the PCFG Parser included in CoreNLP [9] is relatively archaic compared to the rest of the package, and proved too slow for this application. While CoreNLP does support a newer, neural network parser [4], we did not experiment further with that.

3.3.1.2   Berkeley Neural Parser

Berkeley's neural network parser [8] (Benepar) uses neural networks and runs in near-linear time. Generating a parse is a simple function call in python:

```python
parser = spacy.load('en_core_web_md')
parser.add_pipe("benepar", config={"model": "benepar_en3"})
# ...
parse = parser("Benepar can parse untokenized text consisting of
                multiple sentences")
```

Being able to parse a raw string consisting of many sentences is a requirement for this application. However, in our testing Benepar did not always succeed in splitting the text into sentences, as seen in fig. 3.2. Note that the parse in the figure is gramatically correct, but less probable.

3.3.1.3   CRF Parser

We ultimately settled on using the reference implementation for the *SuPar* parser. [14] It outperforms Benepar [8] in terms of parsing speed and accuracy. The
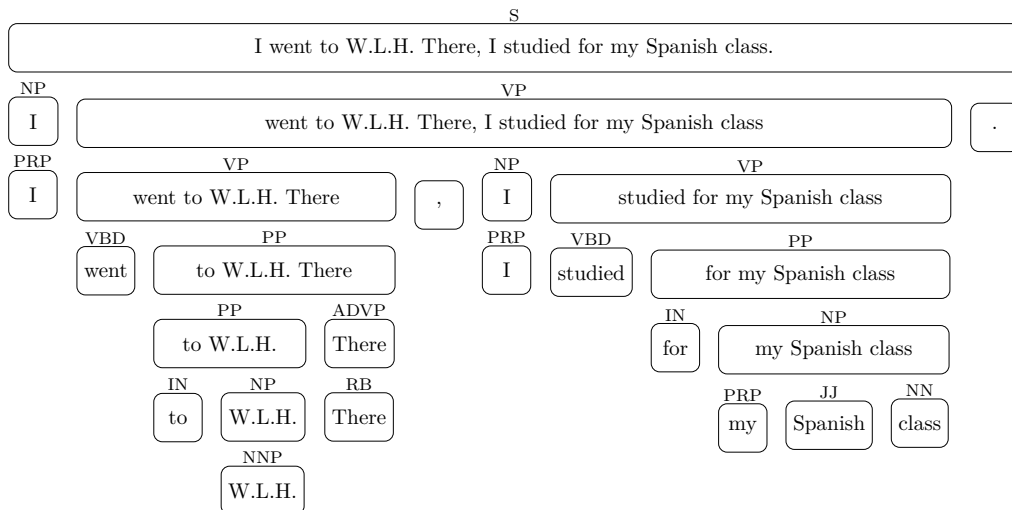
S

I went to W.L.H. There, I studied for my Spanish class.

NP: I

VP: went to W.L.H. There, I studied for my Spanish class

.

PRP: I

VP: went to W.L.H. There

,

NP: I

VP: studied for my Spanish class

VBD: went

PP: to W.L.H. There

PRP: I

VBD: studied

PP: for my Spanish class

PP: to W.L.H.

ADVP: There

IN: for

NP: my Spanish class

IN: to

NP: W.L.H.

RB: There

PRP: my

JJ: Spanish

NN: class

NNP: W.L.H.

Figure 3.2: BENEPAR's parse for *"I went to W.L.H. There, I studied..."*. Note how *"W.L.H. There"* gets parsed as one entity, which breaks sentence segmentation.

SuPar parser does however not accept raw text input, and as such, we need to tokenize the text and split it into sentences. That task is easily accomplished using NLTK:

```
import nltk, supar
parser = supar.Parser.load('crf-con-en')
sents  = nltk.sent_tokenize('The CRF parser needs pre-tokenized input.')
tokens = [nltk.word_tokenize(x) for x in sents]
parses = parser.predict(tokens).sentences
```

The combination of NLTK and SuPar produces the output we expect, as seen in fig. 3.3.

3.3.2 Parse postprocessing

Sometimes, there are multiple correct parses for a single input, and it is useful to convert the parse tree to a standartized form that is more intuitive for the user. We observed that when listing items composed of noun phrases, provided that the noun phrases have a certain number of descendant constituents, the CRF parser groups every few noun phrases as one, producing deeply nested trees as shown in fig. 3.4a.

3.4 Interfacing with the parser

The backend implementation is located in `Sources/Backend`.
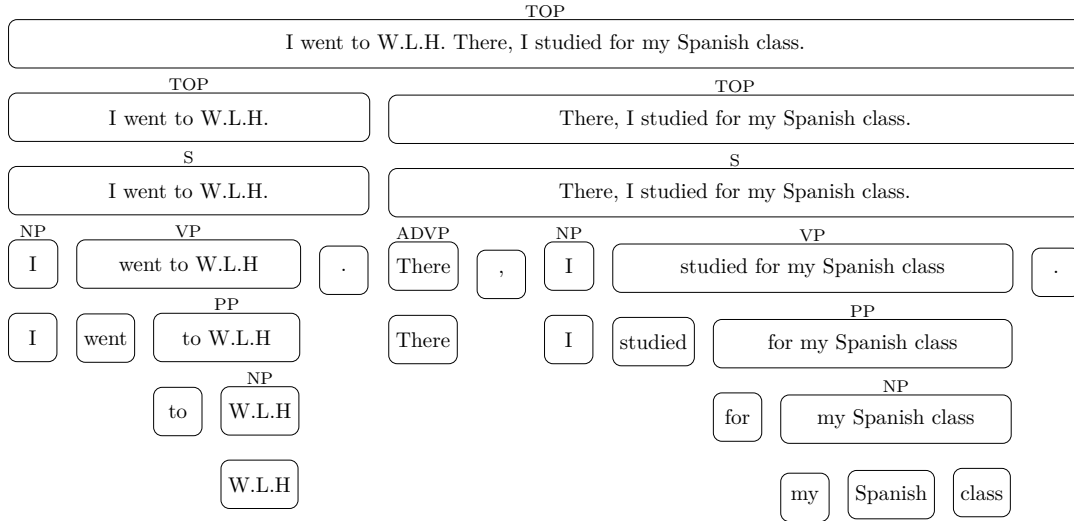
Figure 3.3: SuPar's parse for *"I went to W.L.H. There, I studied. . . "*.

(a) Parser output

(b) Flattened tree after processing
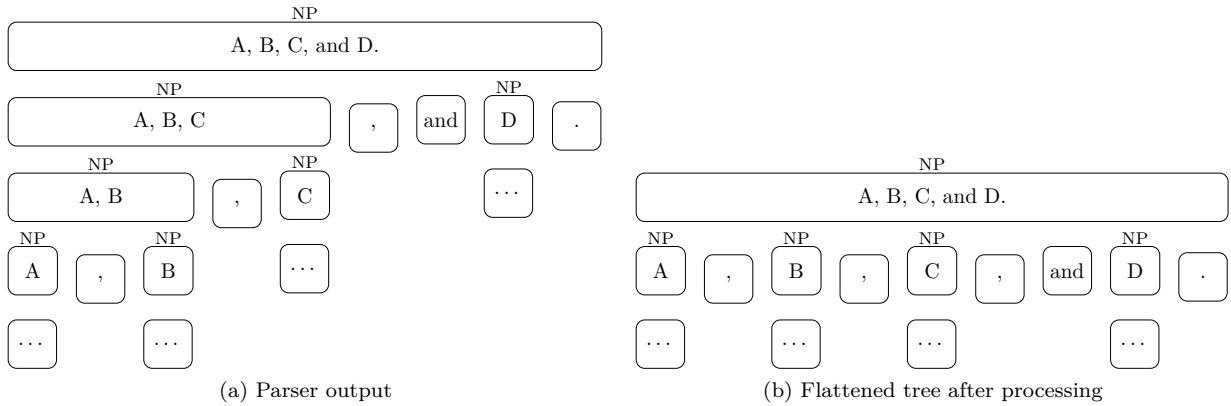
Figure 3.4: A sentence where *"A, B, C, and D"* gets parsed as *"((A, B), C), and D"*

In order to facilitate communication between Swift and the `SuPar` Python package, we use the `PythonKit` swift package [10] to dynamically load a Python interpreter. Then, we use the interpreter to load the Python module at `Sources/Backend/supar_bridge.py`, which imports the `SuPar` parser and wraps it into a function that's easy for Swift to understand.

The frontend waits for changes in the text and parses it in a Combine [2] pipe:

```
NotificationCenter.default
    // Listen for text changes
```

```swift
.publisher(for: NSText.didChangeNotification, object: self)
// Wait so the language server is not overwhelmed
.debounce(for: .milliseconds(50), scheduler: DispatchQueue.main)
.compactMap { ($0.object as? NSText)?.string }
// Don't parse if the text hasn't changed since last parse
.filter { $0.hashValue != self.parse?.hash }
// Parse
.flatMap { NLPServer.parse($0) }
// Sometimes, we receive parses for outdated text. Checking the hash suffices to
// prevent crashes when highlighting
.filter { self.textStorage?.string.hashValue == $0.hash }
// Update state
.receive(on: DispatchQueue.main)
.sink { _ in
} receiveValue: { tree in
    self.parse = tree
    self.highlight()
}
.store(in: &self.subscription)
```

Chapter 4

User interactions

We implement a few commands for creating selections as described below. All resulting selections shown in this section are based on the text and selection shown in fig. 4.1
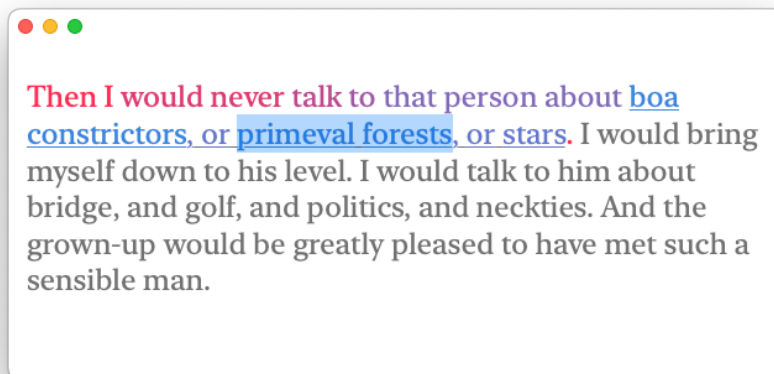
Figure 4.1: Excerpt from Exupéry's "The Little Prince" in our text editor.

- **Expand selection**: Selects the smallest constituent that fully contains the current selection. In this example, expanding would select the noun phrase boa constrictors, or primeval forests, or stars.
- **Refine selection**: Opposite of "Expand selection" – selects a constituent that is a descendant of the currently selected constituent. In the example, that could be primeval or forests. Normally, the most recently selected descendant will be chosen such that this operation negates "Expand selec-

tion". When using the mouse, the descendant closest to the mouse cursor will be chosen.

- **Select enclosing sentence**: Selects the sentence that contains the cursor or the current selection. In the example, this expands the selection to the first sentence: Then I would never talk to that person about boa constrictors, or primeval forests, or stars.
- **Select neighbour**: Selects either of the neighbouring constituents with the same depth. In this example, repeatedly selecting the left neghbour would yield or, ,, boa constrictors.
- **Augment selection**: Appends either of the neighbouring constituents with the same depth to the sellection, or subtracts a children constituent form the selection. In this example, expanding to the right would yield primeval forests,, primeval forests, or, primeval forests, or boa constrictors.

4.1 Keybindings and gestures

Command	Keybinding	Mouse gesture
Expand selection	⌥ + ↑	Pinch out on trakpad, or hold tertiary mouse button and scroll up
Select sentence	⌘ + ⌥ + S	Triple click
Select paragraph		Quadruple click
Refine selection	⌥ + ↓	Pinch in on trackpad, or hold tetriary mouse button and scroll down
Select neighbour	⌥ + ← →	Hold tertiary mouse button and move cursor
Augment selection	⇧ + ⌥ + ← →	Hold ⇧ + ⌥ and click on text

All of the above commands are exposed as keyboard shortcuts as shown it the table above. We chose the ⌥ + → ↓ ← ↑ shortcuts since they are reminiscent of Atom's bindings for treesitter navigation [3].

The shortcuts for augmenting selection ⇧ + ⌥ + ← → function differently depending on the selection's "direction". If a selection was initially made by dragging the mouse left to right, ⇧ + ⌥ + → expands at the right end, and ⇧ + ⌥ + ← trims it at the right end.

A triple click was designated to select the sentence under the cursor. This shortcut is already familiar to users for selecting paragraphs in text fields. The default paragraph selection feature has been reassigned to a quadruple click.

On the trackpad, a "zoom-in" gesture by placing two fingers expands the selection, and a "zoom-out" gesture refines it. These gestures are geneally associated with

making objects larger or smaller, and in this case they are lengthening or shortening the selection.

To access commands with a mouse, the tetriary button must be held down. That button is usually the scroll wheel itself. While the mouse is dragged with the tetriary button, the selection changes to match the constituent directly under the mouse cursor.

Chapter 5

Syntax highlighting

To better inform the user of the structure of the text, we highlight words in different colours depending on how deep they are embedded in a sentence. Observe that in fig. 3.1 the subject of the sentence (*"She"*) is highlighted in orange, as it is embedded close to the sentence root. The objects in the verb phrases are embedded deepest, and are thus highlighted in blue.

Syntax highlighting is not completely foreign to word processors. iA Writer is capable of highlighting parts of speech (POS) in different colours (see fig. 5.1a.) In the author's experience, this sometimes leads to visual clutter and does not intuitively reflect the structure of the sentence. Observe that using our highlighting in fig. 5.1b, the phrases secondary to the meaning are highlighted in a more dull colour, which helps skim-read the sentences as *"I bought oranges, bananas"*. Meanwhile, in fig. 5.1a attention is brought to all verbs: *"bought"*, *"will use"*, *"make"*, *"make"*.

(a) POS highlighting in *iA Writer*

(b) Depth highlighting in our editor

Figure 5.1: A same sentence highlighted in iA Writer and our editor

To reduce clutter, highlighting is only displayed for the currently selected sentence. Moreover, paragraphs other than the one containing the cursor are faded out to bring focus to the cursor, akin to *iA Writer*'s "Focus" feature.

To help the user make selections, we underline the constituent that contains the user selection. The underlined text is what would get selected with the "Expand selection" command.

We provide a few colour schemes to choose from in the View ⟩⟩ Colourschemes menu. Additional colour schemes can be defined in the `colorSchemes` dictionary in `Sources/Frontend/Highlighting.swift`, and they will be automatically listed in the menu.

Chapter 6

Results and Future Work

We developed a simple word processor with additional interactions for selecting common phrases in natural text. Executable source code for the project is available at https://github.com/hristost/CPSC490-SemanticSelections-Implementation [1]. Consult the `README` file in the repository for instructions on how to run.

We presented the finished word processor to five users, and gathered the following verbal feedback:

- The triple-click gesture to select a sentence is particularly useful.
- The multitouch trackpad gestures are intuitive, and users could see themselves using that selection method after more practice.
- The phrases selected by our word processor are useful for editing and copying text.
- One user said that the syntax highlighting brings attention to subjects and important parts of sentences, and another user commented that they were able to skim text faster than usual. The latter also concurred that the colours made it easy to spot superfluous passages that could be redacted while editing.
- When comparing to iA Writer, one user found iA Writer's POS highlighting more natural, while the others sided with our proposed constituent highlighting. Nevertheless, one of them shared that although they find POS highlighting cluttered and too granular, it might be useful to have that information in some cases.

---

[1] Additionally, the Markdown / LaTeX source for this PDF can be found at https://github.com/hristost/CPSC490-SemanticSelections-Report

6.1 Future work

There is a lot more work to be done this approach to be usable in everyday work. Firstly, more accurate parsing and preprocessing is needed. In the sample text bundled with the source code, NLTK mistakengly merges two sentences as one when the former ends in a quotemark. On occassions, SuPar would produce possible but highly improbable parses. In the following sentence, the highlighted text denotes a complete constituent according to SuPar:

> I bought oranges that I will use to make juice, bananas that I will use for banana bread, and milk.

Although this parse is not incorrect per se, (it is grammatically possible that we use bananas to make milk,) it is unlikely, especially given the Oxford comma before "and". Prepending *"some"* before *"milk"* fixes the parsing.

It is worth gathering more data on such edge cases, and researching which parsers fare best.

6.1.1 Incremental parsing

Currently, text is re-parsed on every edit, which leads to a perceivable lag with longer passages. We can instead choose to only parse the sentences being edited, and merge the result in the existing parse tree.

6.1.2 Language support

This type of work should not be restricted but just English language text. As long as a constituency parser exists for a given language, this word processor can be modified to work with it.

Currently, the word processor only supports English text. Very limited support for Chinese is implemented thanks to SuPar's Chinese constituency parsing model. Parsing in Chinese can be activated using the [Edit ≫ Languages ≫ Chinese] menu. Currently, it uses the *JieBa* python package [5] to tokenize text, and is limited to parsing a single sentence.

We found that the proposed interactions and text highlighting work just as well as in English text, as shown in **??**. However, the combination of the JieBa tokenizer and SupPar's model proved unreliable. Substituting the last word "精神" for "神奇" yields a very different parse structure, even though both words are used as noun phrases in this context.

6.1.3 Tokenizing and parsing of structured text

Sometimes, natural language text has bits of structured text in it, such as URLs and phone numbers. Instead of these being represented as leaf nodes in the syntax tree, they can be tokenized and parsed separately to allow the user to easily select parts of them.

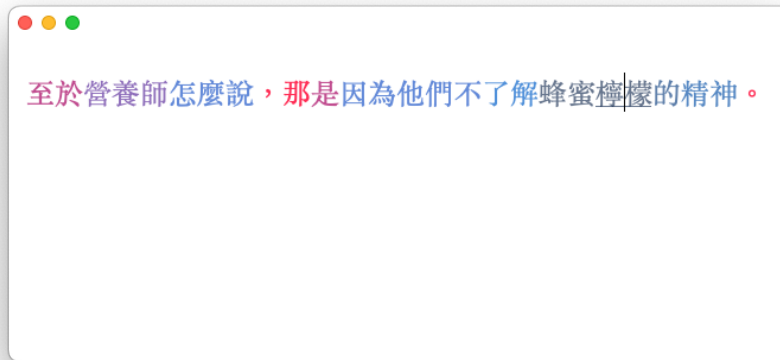Figure 6.1: The word processor highlighting a sentence in Chinese

6.1.4 Semantic selection in all text fields

Lastly, some of the interactions researched in this project, such as selecting complete sentences, are good to have in any text field presented to the user. It would be interesting to explore if it is possible to extend the text field component in a Linux GUI to have such behaviour.

Bibliography

[1]   *Apple Developer Documentation*. selectionGranularity. URL: https : / / developer . apple . com / documentation / appkit / nstextview / 1449165 - selectiongranularity.

[2]   *Apple Developer Documentation*. Combine. URL: https://developer.apple. com/documentation/combine.

[3]   *Atom understands your code better than ever before*. Oct. 2018. URL: https://github.blog/2018-10-31-atoms-new-parsing-system/ (visited on 04/09/2021).

[4]   John Bauer. *Shift-Reduce Constituency Parser*. Accessed April 2021. URL: https://nlp.stanford.edu/software/srparser.html.

[5]   *fxsjy/jieba on Github*. Feb. 2020. URL: https://github.com/fxsjy/jieba (visited on 03/24/2021).

[6]   *hristost/SwiftCoreNLP on Github*. Mar. 2021. URL: https://github.com/ hristost/SwiftCoreNLP (visited on 04/10/2021).

[7]   *Kakoune Documentation*. Object Selection, Inner object. URL: https:// github.com/mawww/kakoune/blob/master/doc/pages/keys.asciidoc# inner-object.

[8]   Nikita Kitaev, Steven Cao, and Dan Klein. "Multilingual constituency parsing with self-attention and pre-training". In: *arXiv preprint arXiv:1812.11760* (2018).

[9]   Dan Klein and Christopher D Manning. "Accurate unlexicalized parsing". In: *Proceedings of the 41st annual meeting of the association for computational linguistics*. 2003, pp. 423–430.

[10]   *pvieto/PythonKit on Github*. Mar. 2021. URL: https://github.com/pvieito/ PythonKit (visited on 03/24/2021).

[11]   T.W. Stewart, N. Vaillette, and Ohio State University. Department of Linguisticsh. *Language Files: Materials for an Introduction to Language & Linguistics*. Ohio State University Press, 2001. ISBN: 9780814250761. URL: https://books.google.com/books?id=9KQlAQAAIAAJ.

[12]   Larry Tesler. "A personal history of modeless text editing and cut/copy-paste". In: *Interactions* 19.4 (July 2012), pp. 70–75. ISSN: 1072-5520. DOI: 10.1145/2212877.2212896. URL: https://doi.org/10.1145/2212877.2212896.

[13]   *VIM Reference Manual*. Cursor-motions navigation, Text Object Selection. URL: https://vimhelp.org/motion.txt.html#object-select.

[14] Yu Zhang, Houquan Zhou, and Zhenghua Li. "Fast and Accurate Neural CRF Constituency Parsing". In: *arXiv preprint arXiv:2008.03736* (2020).