CAREER (Comprehensive Automated Resume & Employment Experience Renderer) Language Specification

Harry Albert

May 3, 2024

Introduction

Creating a beautiful resume is often tedious and time consuming, but this annoying task is unfortunately extremely important. Making your resume look professional and easily readable can be the difference between a recruiter considering you for a job and throwing your application away. Formatting a resume well entails taking the time to organize your experience, skills, and interests in an aesthetically pleasing way using an often uncooperative word editing software. This means that making a resume can be both time consumign and painful, while still mandatory for career goals.

CAREER solves this issue by automatically formatting your information for you, taking as input some simple text and outputting a well designed resume. CAREER will always output a visually pleasing and readable product, without the fuss and frustration of actually formatting everything yourself. This also allows for the easy modification of one's resume without the need to go back and reformat everything. In summary, CAREER saves time by making resume creation a breeze, democratizing quality resume creation by putting a well formatted resume at anyone's fingertips.

Design Principles

The main goal of CAREER's design is to be easily readable and modifiable by the user. Any user who has skimmed over CAREER's documentation should be able to easily read and understand a resume written in CAREER. Users should be able to go back to their resume document after not looking at it for a year and quickly find and update whatever has changed in that time. This means that all of CAREER's commands and functions should be easily readable and understandable, ideally with little reference to the documentation. The other main design principle guiding CAREER is conciceness. One of the problems CAREER is solving is that resume writing is tedious and time consuming. Requiring the user to write lots of boiler plate code to create a CAREER resume would not fix this problem.

Examples

Example 1:

This example program shows how one could create a complete, basic resume using CAREER. This resume includes a title, contact information, and multiple unique sections. Each of these sections has multipe titles with different alignments, as well as bullet point items. While this resume is slightly long, it is easily readable and modifiable. It is easy to imagine how a resulting resume would look based on this code.

```
*HEADER Harry Albert
*SUBHEADER (111)111-1111
*SUBHEADER hsa1@williams.edu

*EDUCATION
*TITLE_LEFT Williamstown, MA
*TITLE_CENTER Williams College
*TITLE_RIGHT Fall 2021 - May 2025
```

*ITEM Teaching Assistant for introductory and upper-level CS courses
*ITEM Activities include PAC Comedy Club, Opinions Columnist for the Williams Record Newspaper

*EMPLOYMENT
*TITLE_LEFT Software Engineer, Intern
*TITLE_CENTER Bubble
*TITLE_RIGHT Summer 2023
*ITEM Bubble is a no-code web development platform. I worked on the Growth Team
*ITEM Contributed to multiple successful experiments, including improving the editor UI and user journey

---

Example 2:

This example program is similar to the previous program. However, it adds on the idea of generic sections. Having built in commands for common sections like employment and education is great, but there should still be room for the user to create some of their own generic sections. This idea of generic types can and should be extended to other aspects of the CAREER language.

*HEADER Harry Albert
*SUBHEADER (111)111-1111
*SUBHEADER hsa1@williams.edu

*SECTION Languages & Technologies
*ITEM Javascript/Typescript, Python, HTML + CSS, F#, Java, C
*ITEM React, Scikit-Learn, PyTorch, SolidJS, REST APIs

Example 3:

This example program introduces the idea of brackets as limiters on the skoper of modifiers. Only the words within the brackets are bolded under the projects section.

*HEADER Harry Albert
*SUBHEADER (111)111-1111
*SUBHEADER hsa1@williams.edu
\\ TODO: add in website link here
*SECTION Languages & Technologies
*ITEM Javascript/Typescript, Python, HTML + CSS, Java, C \\ remember to add on F# when I'm done with PL
*ITEM React, Scikit-Learn, PyTorch, SolidJS, REST APIs

*SECTION Projects
*ITEM *BOLD [ML Lung Cancer Prediction Tool]: Created a neural network to predict lung cancer based on several easily self-diagnosable conditions. Published website where users can interact with this model to democratize access and allow users to play with the model. \\ Remember to bring this up in job interviews as an example of ML experience
*ITEM *BOLD [Booktrak]: Independently designed and created Booktrak for WSO (student run website for Williams College). Booktrak is a platform where Williams students can buy and sell used books, and link their book listings to specific classes in order to better facilitate used book sales.

Language Concepts

The core concept of this language is that there are modifiers and text. Text is a primitive, and it is defined as any string that is not a modifier. For example, in the line ”HEADER Harry Albert,” ”Harry Albert” is the text. Modifiers are (conventionally all caps) functions, indiciated by *, that change how text is displayed. For example, the SECTION modifier bolds the text that follows it and adds a horizontal line above it to indicate a new section. Ideally, users will be able to create their own modifiers to make the language more groweable. The ITEM function is a special modifier function that can only be applied to the start of a line. Finally, modifiers can be stacked. For example, ITEM creates a bullet pointed text, and BOLD (used within an item) creates a bolded piece of text. When brackets are included after a modifier, the modifier will apply only to the text included in the brackets. If brackets are not included, the modifier will apply to all text until a new line.

Formal Syntax

<expression> ::= <line><expression>*
<line> ::= <line_content> \n
<line_content> ::= *LINE␣<formatted_text>*
| *SUBSECTION_TITLE ␣<subsection_content>*
| <formatted_text>*
<subsection_content> ::= [<string>] | [<string>][<string>] | [<string>][<string>][<string>]
<formatted_text> ::= <modifier> | <string>
<modifier> ::= *<mod_fun>␣[<formatted_text>*] | <mod_fun>␣<formatted_text>*
<mod_fun> ::= HEADER
| SUBHEADER
| SECTION
| TITLE_CENTER
| TITLE_LEFT
| TITLE_RIGHT
| ITEM
| BOLD
| UNDERLINE

<string> ::= <str_char>*
<str_char> ::= $c \in \{\text{all characters}\} \wedge c \notin \{\backslash n, *, [,]\}$

Semantics

CAREER is essentially composed of many lines of strings. Thus, the primitive form in CAREER can be thought of as either a character or a string (a list of characters), depending on how deep you want to go into the program implementation. Basic strings can only be composed of numbers, letters, spaces, and punctuation (this is a potential shortcoming that could be improved in the future). Modifier function strings (which are used as commands for text formatting) can have any character. However, all of these function strings are defined within the definition of CAREER itself, instead of in an actual CAREER program. Modifier function strings are all uppercase and snake case by convention.

The combining forms for modifier strings and basic strings are modifiers, lines, and expressions. Modifiers combine modifying functions and other text (other text could include strings or text that has already had some different modifying function applied to it, or both). The left side of a modifier is a modifier function string such as ”*SUBHEADER” or ”*BOLD”. The right side of a modifier is either another modifier or a basic string, or both. Modifier bodies extend as far right as possible. This means that, in the example ”*ITEM *BOLD hello world”, the BOLD command would apply to the entire string ”hello world”. If the user wants BOLD to only apply to a part of the string, they can use quotes to limit the scope of the function. Returning to the previous example, this would look like ’*ITEM *BOLD ”hello” world’, a command which would result in only ”hello” being bolded. The other combining form in CAREER

is a line.

A line is simply a list of modifiers and strings, and each line corresponds with an actual line from the input and outputs an actual formatted line in the resluting resume. The result of evaluating '*ITEM *BOLD [hello] world' would be Line([Modifier ("ITEM", [Modifier ("BOLD", [String "hello"]); String " world"])]). Notice that the modifier "BOLD" only applies to the string "hello", as hello is surrounded by brackets, but the modifier "ITEM" applies to all parts of the string. Finally, an expression is simply a list of lines. Each AST will only contain one expression which contains a list of all of the lines in a resume.

CAREER will take the path to a text file as input, and will first transform that text file into an AST. Evaluating the AST should result in a latex file that, when built, outputs a well formatted resume. There will be a command that, when including in the running of a CAREER program, will automatically compile the resulting latex file into a pdf. Each string can be fed into the latex file exactly as it is. Each modifier will be transformed into a latex command that wraps the modifiers inner text, formatting that text in the desired fashion. Each line will combine all of the resulting latex-formatted strings from each of its inner modules and strings. Finally, the expression will simply combine all of the latex-formatted line results into one complete resume. This resume (which is just my resume, I'm planning on making this anonymous later) is an example of what a CAREER output could look like (based on the first program example).