Using Trees for Efficient and Correct Parsing of Command Inputs

Alexander Söderberg
contact@alexander-soderberg.com

ABSTRACT

This paper discusses how a tree structure with certain strict requirements would provide a suitable way to store commands for use in computer software. The paper outlines a way to represent commands using fragments called command components, and a method of storing these fragments that allow for unambiguous parsing of user input into pre-defined command structures. Furthermore, attempts are made of processing the tree elements to reduce time complexity when possible.

1 INTRODUCTION

Commands are commonplace in a variety of computer software, especially command-line interface (CLI) based applications and video game terminals. Essentially, a command is an input string supplied by the user of the program that prompts an action. These strings are often hierarchial, where each segment of the string provides additional context. Such hierarchical commands will be the topic of this discussion.

These hiarchical commands often start with a prefix (often denoted "the command") followed by additional command arguments, like such: <command> <argument 1> <argument 2> ... <argument n>. Sometimes not all arguments are required for the command to function. Such arguments that can be omitted are called *optional arguments*. In this text, optional arguments are written as [argument] whereas required arguments are written as <argument>. When an argument is not variable, but instead a fixed string, no surrounding brackets will be used.

When using such commands in computer software, many different problems may arise. Some common problems include: end-user usability, parsing ambiguity, efficiency and code complexity. In particular, the method used to store and query the commands internally will have a significant impact on the usability and maintainability of the code base. This paper will explore one particular tree-structure and how it can be used to avoid problems with command parsing (such as ambiguity) whilst allowing the software to provide the user with verbose information, so that they may correct any mistakes in their input strings.

2 COMMAND STRUCTURE

2.1 Command Components

Commands consist out of a unique path of minor command components. As mentioned in the introduction, these components may be required or optional, and may be variable, or fixed strings. An example command would be the UNIX command *cp* which takes in a source, and a destination (ignoring potential command flags). The command structure could thus be written as cp <source> <dest>. Thus, it has three components:

(1) *cp*: A fixed string
(2) <*source*>: A required variable
(3) <*dest*>: A required variable

Formally, a command is a 2-tuple (A, C) where A is a set of aliases $A = \{a_1, \ldots ...a_n\}$ where a_1 is the main command string. Aliases are fixed string command components that initiate a command chain. C is a sequence of command components $< c_1, \ldots, c_n >$ and will be referred to as "the component chain". In all cases, $a_1 = c_1$. That is, the main command aliases is the initating command component in the component chain.

2.2 Variable Components

Variable components are associated with some type of parser, that parses the string input into a pre-defined data type. The only restriction on these data types is that they must necessarily be string serializable.

2.3 Equality

Two command components are identical iff their name and parsers are identical.

2.4 Command Handlers

In order for bind functionality to the command, the last command components in a command chain will thus contain a pointer to a command handler. All other command components may not contain such a pointer.

Let $H(c)$ be a function that returns the command handler associated with a command component. Thus, for a component chain C with length n, it must follow that $H(c_1) = H(c_2) = \cdots = H(c_{n-1}) =$ null and $H(c_n) \neq$ null.

The contents of the command handler is not important for this discussion, and will thus be omitted.

3 INPUT

Command input will be in the form of a last-in first-out (LIFO) queue, consisting of strings. This queue will be popped by the command components as the input is being parsed into a command chain.

When parsed by a variable command components, the parsed input will be stored in some kind of key-value map, where the key is the name of the variable component.

4 COMMAND TREE

All commands may be stored in a tree structure. This is done by allowing each node in there to be a command component, and letting each leaf node contain a pointer to a command handler. If this is the case, then each path from the root to a leaf node will represent a unique chain of command components. If the top node

is a command alias, then such a path would completely represent a command.

When parsing the user input, the command tree will be walked until one of four scenarios occur:

(1) The input queue is empty at a non-leaf node
(2) The input queue is empty following a leaf node
(3) The input queue is not empty following a leaf node
(4) No child node is able to accept the input

In the first scenario, the user should be informed that the command input does not correspond to a complete command chain. The user could be shown the syntax of the complete command chain, if it is able to be determeined unambiguously.

In the second scenario, a complete chain has been parsed and the command handler may be called. This is a successful case.

In the third case, the user has supplied too many arguments. The user should be informed about this.

In the fourth case, the user input does not correspond to any command chains. The user should be informed about this.

Only the second scenario will be discussed in this paper. How the other scenarios are handled is up to the implementing application.

4.1 Ambiguity

If two children from a node are variable, then it will not be possible to determine which child node should be allowed to consume the input. Because of this, no such cases may be allowed. Furthermore, it cannot be determined whether a fixed string component should be allowed to consume the input or not, in the presence of a variable component. More specifically, if $C(c)$ is a function that returns a set containing all child nodes of a command component node, then $(\exists k \in C(c))(k \text{ is variable}) \Leftrightarrow |C(c)| = 1$.

In the case that an optional command component is present, it may not be followed by a required component. Allowing such a case would always introduce a level of ambiguity. It is therefore crucial that if $c_i \in C$ is variable for some command, then it must be true that for $C^* = \{c_1, \ldots, c_{i-1}\}$, $(\forall k \in C^*)(k \text{ is required})$.

4.2 Populating the tree

When adding a command $K = (A, C)$ to the command tree, the elements in C will be considered individually, as such:

```
1. let n be head of tree
2. ∀c ∈ C:
3.   if c ∉ C(n):
4.     add c to C(n)
5.   if |C(n) > 1|:
6.     sort C(n) lexiographically
7.   let n = c
8. set H(C_|C|) to the command handler of the command
```

The sorting step (line 5) allows the command parser to perform a binary search on the node children during the parsing process. If a node only has one child, its only child must already be in lexiographical order.

4.3 Parsing Input

Parsing the input consists of walking the command tree until one of the scenarios in Section 4 occur.

The input queue will be forwarded to each command component. If the command component is a fixed string, it will be popped iff the head of the queue corresponds to the required string. If the component is instead a variable component, the queue will be forwarded to the component parser.

Because the children of a node are always sorted in lexiographical order, binary search may be performed to determine whether or not the input queue corresponds to a command component. Thus, a linear search of all node children can be avoided.

If after parsing, the path ends at a leaf node and the input queue is empty, a unique path exists that must necessarily correspond to a command, and the command handler may be called.

4.3.1 Pathological Case: Single Path. If the command tree consists of a single path of fixed strings, walking the path would be the same as walking a linked list. As such, the time complexity would be $O(n)$.

If all components in the path are instead combined into a single component with a string made from concatenating all the invidual fixed strings, this entire path could be determined in $O(1)$ time instead.

Furthermore, all consequtive command components c_i, c_j in a chain may be compacted into a concatenated string, as long as $C(c_i) = \{c_j\}$, given that c_i and c_j are both fixed string components.

5 CONCLUSION

The above discussion shows that a collection of commands may be stored in a tree to then be unambiguously parsed from user input, given that they follow a set of strict rules regarding the command structure.