Command Parser
Last modified: 2018-07-23

Overview

The command parser for the *Cetus* game must accomplish two goals. First, it must determine if a command entered by the user is grammatically correct and able to be parsed. Second, it must identity parts of the command and parse out the important words necessary for the game to take action. Thus, to achieve this, a context-free grammar must be defined. This will then act as the rule set the parser uses to determine if a command is valid or not. Then, an algorithm must be defined that can both verify and extract words from a command. The verify and extract operations can happen within the same algorithm as the parser must know the type of word it is examining to determine if the sentence as a whole is valid, thus when an important word is detected, it can be extracted as the parser continues.

Context-Free Grammar

A context-free grammar is a grammar that can define all possible strings in a given language. A formal definition of a grammar is necessary for the parser to determine correct commands entered by the user. For *Cetus*, a relatively small grammar will be defined. A context-free grammar, *G*, is defined as such

$G = (N, \Sigma, R, S)$, where:
N is a set of non-terminal symbols
Σ is a set of terminal symbols
R is a set of rules of the form $X \rightarrow Y_1Y_2...Y_n$ for $n >= 0$, where $X \epsilon N$, $Y_i \epsilon (N$ or $\Sigma)$
$S \epsilon N$ is a distinguished start symbol

Therefore, the grammar for *Cetus* is defined as follows:

$N = \{ C, VP, NP, PP, N' \}$
$\Sigma = \{ ADJ, ART, N, P, V, \}$
$R =$

C → VP

VP → V
VP → V NP
VP → VP PP

PP → P NP

NP → N'
NP → ART N'

N' → N
N' → ADJ N'

$S = C$

Note: C = command, VP = verb phrase, NP = noun phrase, PP = prepositional phrase, N = noun, ADJ = adjective, V = verb, ART = article, P = preposition

Parsing Algorithm

The parsing algorithm will first iterate over the command given by the user and compare each word against a dictionary of words to determine the proper tag to give the word from category *Σ* of the grammar. From there, it continues to reduce tags together until the highest level tag, *C,* is produced. If a *C* is produced, then the command issued by the user is grammatically correct. Otherwise, the command is not grammatically correct. In the following pseudocode, assume the *reduce()* function reduces one or two tags to their higher level tag according to the grammar, assume *reduceStack(),* reduces the tags in a stack as much as they can be, and assumed *canBeReduced()* and *stackCanBeReduced()* functions return a Boolean indicating if a reduction operation can occur.

```
ParseCommand( Sentence ):
    let fA be an empty array that will store features/objects
    let v be an empty variable that will store the verb the user entered
    let d be an empty variable that will store the direction the user wishes to go
    for each word in Sentence:
        tag each word according to  Σ;
        push tag onto stack W

    while ( top(W) != C ):
        if W.length > 1:
            w1 = pop(W)
            w2 = pop(W)

            // add any features to features array
            if w1 == F':
                fA.add(w1)
            if w2 == F':
                fA.add(w2)


            // store verb that user entered
            if w1 == V:
                v = w1
            else if w2 == V:
                v =w2

            // if words one and two represent a direction and a movement, return
            if w1 == D && w2 == M:
                d = w1
                return (true, d)

            // otherwise, we need to keep parsing the command
            if canBeReduced(w1, w2):
                push(W, reduce(w1, w2))
            else:
                push(W, w2)

                if canBeReduced(w1):
```

```
                    push(W, reduce(w1))
                else if stackCanBeReduced(W):
                    push(reduceStack(W), w1)
                else
                    return false
        else:
            w1 = pop(W)
            if canBeReduced(w1):
                push(W, reduce(w1))
            else:
                return false
    return (true, v, fA) // return verb and features to be acted upon
```

Data Structures

The command parser will use a hashtable as a dictionary with a one-to-one lookup mapping words to their associated tags according to the grammar. There will be one tag not associated with the grammar that indicates an invalid word. At the start of the *Cetus* game a dictionary will be loaded into memory for the command parser to use.

The command parser will use a stack for managing tags in the parsing algorithm.

The command parser will return to the game engine a data structure that contains the verb the user entered and the parameter verb is acting upon.

```
struct parsedCommand {
    char[] verb;
    char[] param;
}
```

API

To call the command parser from the game engine, the engine will pass a pointer to a *parsedCommand* struct where the parser can fill in the necessary information, a string that is the command the user entered, and a pointer to a hashtable where the command parser can perform tag lookups for words. The function will return a Boolean value indicating whether the command entered by the user is grammatically and syntactically correct.

*int parseCommand(struct parsedCommand *parsedCommand, char *commandString, struct hashTable *dictionary)*