Transition-Based Dependency Parsing

Introduction

For this assignment, you must implement a transition-based dependency parser as discussed in class and also in a conference paper that in included in the distribution for this assignment. As an essential part of the assignment, you must also develop a primitive classifier based in a particular manner based on a furnished dependency corpus. This classifier will serve as the oracle for your parser. Your oracle must also include some special case rules that are discussed further below.

You can pick up the complete assignment distribution here. The distribution contains the training corpus, two sets of sample inputs with associated outputs, and the conference paper mentioned above.

Program input will be a text file that will contain a POS-tagged sentence. The name of the file will be furnished to your program as a command-line argument. It will be the job of your parser to generate the output that represents a dependency parse of that sentence for the particular oracle that we have asked you to build.

The sections below describe the specific requirements that your program must satisfy.

Please note that you must develop your own program. This is not a group assignment. The instructor reserves the right to submit your program to a software similarity detector.

Programming Language

You may use any of the following programming languages: C, C++, Java, or Python. It is the specific intent of this assignment that you build your classifier and parser yourself using the basic general purpose programming constructs of your chosen programming language. Therefore, you may not use any pre-built dependency parsers or classifier library modules. Any use of unauthorized libraries will result in substantial grade deductions.

Your program must run on the instructor's machine, which contains only standard distributions for gcc, g++, Java 8, and Python 3.6.2. You must not require the instructor to install any additional libraries or extensions in order to run your program.

Command Arguments

Your program must accept one command arguments representing the name of the text file that contains the input test sentence to parse. The corpus will be the same for all test runs, so you may hard code the corpus file name ('wsj-dep.txt').

It is not necessary for your program to do any validation of either the number of arguments or the presence of the named file. All invocations of your program will be well formed.

Input Test File Format

Each input test file will contain exactly one POS-tagged sentence. Each line will contain a separate token, followed by the forward-slash character ("/") and the POS tag for the token. See the following sample input files that are included with the distribution: piper.txt and hearing.txt. A sample appears below:

```
Peter/NNP
Piper/NNP
picked/VBD
a/DT
peck/NN
of/IN
pickled/JJ
peppers/NNS
./.
```

Corpus File Format

The corpus file will be a numbered, pre-tokenized text file, with one token on each line. Also on each line will be the index number for the token, the POS tag for the token, and the index number of the head associated with the token. A head value of 0 indicates an arc from the root. Index numbers start with the number 1 for each sentence in the corpus. A sample sentence from the corpus follows:

```
1 In    IN      14
2 composite     JJ      3
3 trading       NN      1
4 on    IN      3
5 the   DT      9
6 New   NNP     9
7 York  NNP     9
8 Stock NNP     9
9 Exchange      NNP     4
10 yesterday    NN      14
11 ,    ,       14
12 Upjohn       NNP     13
13 shares       NNS     14
14 rose VBD     0
15 87.5 CD      16
16 cents        NNS     14
17 to   TO      14
18 $    $       19
19 38.875       CD      17
20 apiece       RB      19
21 .    .       14
```

Sentences in the corpus are separated by empty lines.

The corpus we are using ('wsj-dep.txt') is an abridged version of a publicly available subset of the Penn Treebank corpus that has been annotated for dependency parsing.

Unfortunately, the corpus does not contain dependency relation information, so your parser will need to determine only the correct transition arcs for the parse.

Developing the Oracle

For a transition-based dependency parser, the oracle performs two functions. First, it determines whether a particular transition is permitted for two given words. And second, where multiple transitions a permitted for two given words, it determines which of all possible transitions should be taken by the parser.

The oracle you must develop for this assignment will be a very simple one that bases its decisions on only the following four items of information: (i) the POS tag for the second token in the stack; (ii) the POS tag for the top token in the stack; (iii) the current size of the stack; and (iv) the current size of the buffer. The stack and buffer are the two data structures that are used by the parser, as discussed in lecture and in the following conference paper that is included in this assignment distribution: "Non-Projective Dependency Parsing in Expected Linear Time", by Joakim Nivre (2009).

You must process the training corpus to determine which transitions are possible for the tokens at the top (the j-th position) and in second place (the i-th position) in the stack. For situations where both right and left arc transitions are possible, your oracle must choose the direction of the arc that is more prevalent for the particular combination

of tags for the i-th and j-th positions. If neither "Left-Arc" nor "Right-Arc" action is permitted, then your oracle must return the "SHIFT" action as the default action.

Special Oracle Rules

The section above presents the basic oracle functionality. Your oracle must also implement the following rules, in the order listed here, prior to applying the basic action determination above.

The special oracle rules are as follows:

1. If the first character of the tag for the i-th element is "V" and the first character of the tag for the j-th element is either "." or "R", then your oracle should return "Right-Arc" as the action for the parser to take;
2. Else if the current size of the stack is greater than two (not including the ROOT node) and the first character of the tag of the i-th element is "I" (upper case I) and the first character of the tag of the j-th element is ".", then the oracle should return "SWAP" for the parser action;
3. Else if the buffer is nonempty and the first character of the tag for the i-th element is either "V" or "I" and the first character of the tag for the j-th element is either "D", "I", "J", "P", or "R", then the oracle should return "SHIFT" for the parser action.

If none of the above rules applies, then your oracle should return the action as determined in the previous section.

Please note that the second rule above returns the "SWAP" action, which is defined in the Nivre article. This is the only situation in which your oracle should return that action.

Output Format

All program output should be to the console and should be in the same format as the complete program output in the sample text documents "piper.out" and "hearing.out", which are contained in the assignment distribution.

The required sections of the program output are as follows:

1. Program header identifying UCF and this course, and you as the author of the program.
2. A corpus statistics section, containing the items shown. The "Root-Arcs" count refers to those Right-Arcs that do not involve word-to-word arcs, as these are not included in the Right-Arc counts section, below.
3. A "Left-Arc" array reporting section, that shows the nonzero counts of Left-Arc transitions from words tagged with the indicated tags to words tagged with the tag at the left margin of the line of output.

4. A "Right-Arc" array reporting section, that shows the nonzero counts of Rightt-Arc transitions from words tagged with the tag at the left margin of the line of output to words tagged with the indicated tags.
5. An "Arc Confusion" section showing those tag-tag combinations where both left and right arcs were observed in the corpus. Each entry shows a tag plus the number of Left-Arc transitions observed, then the number of Right-arc transitions observed. This reporting section also includes a statement of the number of such tag-tag combinations.
6. A final section showing the tagged input sentence, followed by the sequence of parsing actions of the dependency parser. Each line of the parsing output shows first the contents of the stack (with its head to the right), then the contents of the buffer (with its head to the left), followed by the action the parser takes given this configuration. The actions must be either SHIFT, SWAP, Left-Arc, or Right-Arc, and for the two arc actions they must indicate which word is the head and which word is the dependent by using an arrow, as shown in the sample output section below. The arrow should start from the head and point to the dependent. This manner of showing the stack and buffer shows the heads of the two structures close together for easy understanding. It follows the format used in the Nivre article. A sample of this output section appears below:

```
Input Sentence:
Peter/NNP Piper/NNP picked/VBD a/DT peck/NN of/IN pickled/JJ peppers/NNS ./.


Parsing Actions and Transitions:

[] [ Peter/NNP, Piper/NNP, picked/VBD, a/DT, peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] SHIFT
[ Peter/NNP] [ Piper/NNP, picked/VBD, a/DT, peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] SHIFT
[ Peter/NNP, Piper/NNP] [ picked/VBD, a/DT, peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] Left-A
[ Piper/NNP] [ picked/VBD, a/DT, peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] SHIFT
[ Piper/NNP, picked/VBD] [ a/DT, peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] Left-Arc: Piper/N
[ picked/VBD] [ a/DT, peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] SHIFT
[ picked/VBD, a/DT] [ peck/NN, of/IN, pickled/JJ, peppers/NNS, ./] SHIFT
[ picked/VBD, a/DT, peck/NN] [ of/IN, pickled/JJ, peppers/NNS, ./] Left-Arc: a/DT <-- peck/NN
[ picked/VBD, peck/NN] [ of/IN, pickled/JJ, peppers/NNS, ./] Right-Arc: picked/VBD --> peck/NN
[ picked/VBD] [ of/IN, pickled/JJ, peppers/NNS, ./] SHIFT
[ picked/VBD, of/IN] [ pickled/JJ, peppers/NNS, ./] SHIFT
[ picked/VBD, of/IN, pickled/JJ] [ peppers/NNS, ./] SHIFT
[ picked/VBD, of/IN, pickled/JJ, peppers/NNS] [ ./] Left-Arc: pickled/JJ <-- peppers/NNS
[ picked/VBD, of/IN, peppers/NNS] [ ./] Right-Arc: of/IN --> peppers/NNS
[ picked/VBD, of/IN] [ ./] SHIFT
[ picked/VBD, of/IN, ./.] [] SWAP
[ picked/VBD, ./.] [ of/I] Right-Arc: picked/VBD --> ./.
[ picked/VBD] [ of/I] SHIFT
[ picked/VBD, of/IN] [] Right-Arc: picked/VBD --> of/IN
[ picked/VBD] [] ROOT --> picked/VBD
```

What to Submit

You must submit your program as a single source file of type .c, .cpp, .java, or .py. Comments at the top of the file should identify this assignment and you as the

author. The comments should also provide complete instructions for compilation (if required) and for running your program using command arguments.