Keigh Rim

Todd Curcuru

For our final project we chose to work on a syntactic parser for Japanese. Japanese is an agglutinative SOV language with relatively free word order. The only inflexible rules about word order/grammaticality are that: verbs come at the end of a clause, except in rare cases of emphasis; and a verb must be present in a sentence for it to be grammatical. But other than the verb (or word + copula), no syntactic categories need be present.

1) “watashi-wa banana-wo tabe-te-i-mas-u”

“I” -- Topic Marker “banana” – Object Marker Verb Stem “eat” + Connector “te” + Progressive

Aux “i” + Polite “mas” + Present Declarative “u”

Talking about me, (I) am eating a banana.

2) “banana-wo watashi-wa tabe-te-i-mas-u”

Given the relatively free word order, this sentence is also grammatical, with the same meaning as above.

3) “tabemashita” = “tabe-mas-(i)ta”

Verb stem “eat” + Polite “mas” + Past “ta”

(I) ate.

For phonological reasons, “i” is epenthesized and “si” -> “shi”

Japanese has explicit case marking in polite speech, where particles are attached to the right of the NP. This greatly aids in the parsing of the free word order before the Verb, and so for our project we assumed that all NPs would have explicit case marking. In casual speech many particles are often dropped, and casual morphemes are attached to the verb. We chose to implement both polite and casual morphemes on verbs, because that did not require much extra work.

The agglutination in morphology is heavily present in the verb, as seen in the examples above, and this is where the bulk of our work ended up, in both the lexicon and the parser.

**Lexicon**

Nouns do not have agreement.

Adjectives and “determiners” are actually syntactic nouns or verbs.

**Parser**

We started work on the parser by first making everything that occurred before the Verb work, since Verbs are a whole problem unto themselves. Luckily, Japanese has no person, gender, or number agreement. So any determiner, adjective, and case-marking particle can occur together with any noun. So we modified the original npRule to no longer check for agreement.

We then needed to be able to parse the case marking particles, so we made functions parsePart for that. And then we decided to call a NP + Particle a Particle Phrase, or PartP for short, since the particle was always necessary. In this partpRule, it needs to parse and match a NP and then multiple particles, since Japanese allows for multiple particles on NPs:

4) “curcuru-sann-ni-mo (tegami wo ka-(i)te-kur-e) ”

Curcuru Vocative Honorific Polite Dative “to” “also”

To Mr. Curcuru as well (write a letter).

However, every particle attached to a NP shares its case feature with the noun, so we needed a way to combine the features on the NP and all particles attached. This function is called superCombine:

superCombine :: ParseTree Cat Cat -> [ParseTree Cat Cat] -> [Agreement]

superCombine cat1 catlist =

concat (map (\x -> combine (t2c cat1) (t2c x)) catlist)

It works by simply taking a category (e.g. NP) and then combining its features with the features of every category in a list (e.g. a list of particles), and then concatenating all those features into one list.

We also changed PPs from Prepositional Phrases to Postpositional phrases, with postpositions coming after the NP they govern.

Next we had to change the original sRule to account for free word order. We did this similarly to how the original vpRule subcat frame worked: we parse for any combination of PartPs, PPs, or VPs, in any order, taking advantage of the many function and disjunctions:

parsePartPorPPorVP :: PARSER Cat Cat

parsePartPorPPorVP = parsePartP <|> parsePP <|> parseVP

parsePartPsorPPsorVP :: [Cat] -> [([ParseTree Cat Cat],[Cat])]

parsePartPsorPPsorVP = many parsePartPorPPorVP

The one caveat with this is that this does not require a VP to be in the sentence at all for it to achieve a licit parse. And moreover it does not require a VP to be sentence finally – it can occur anywhere. We tried many different ways to get sRule to work requiring a VP to end the sentence, but there always seemed to be issues with the implementations – either there were lists when there should not have been, or the sentence would parse correctly but leave out the final VP. We implemented this early in the process and meant to go back to it, but the VP issues were more pressing and ended up taking the remainder of our time.

One benefit of this totally free word order is that it allows for relative clauses without requiring specific handling of it. In Japanese, relative clauses precede the Noun:

5) “watashi no tabe-ta banana (wa oishi-kat-ta des-u)”

“I” Genitive Connector Verb stem “eat” + Past Casual “banana”

The banana that I ate (was delicious).

I imagine it would not be too hard to place inside the NP rule an optional starting VP, but this might run into infinite loops, as happened with Verbs and Auxen (NB: Auxen is abbreviation for Auxiliary Verbs) explained below.

Now for the hard part: the Verbs. Agglutination is awful. We have four types of morphemes for the V, and we labeled them Stem, End, Inf, Fin. Every verb needs at least a Stem and a Fin, which is the end of a Verb. But between the Stem and a Fin there can be other morphemes, and there are dependencies between them – some can go with some but not others. We tried to model this by having the features block unification of combinations that are not licit.

We then changed the original vpRule quite a bit. The first change is that we no longer account for subcategorization. This is an issue, but it would take some serious work to implement. Basically, because there is free word order, and because any and all NPs can be omitted before a Verb, it would require some kind of checking system – where it runs through all NPs in a sentence before the Verb, and checks if their case match one of the allowable ones in a VP’s subcat frame. If there is one that is not allowable, then it crashes.

The second change to the vpRule is to account for agglutinating all the morphemes together.

However we ran into a lot of trouble when we tried to introduce feature replacement:

-

-

-

- Explain here

-

-

- Also explain how Auxen work and why we did not get to them