Parser modification for Japanese support

Project description

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 predicate (including noun + copula), no syntactic categories need be present. Sentence 1) shows a very simple example of agglutination of the Japanese language. Pay attention on verb part, the second half.

1)

watashi	*-wa*	*banana*	*-wo*	*tabe*	*-te*	*-i*	*-mas*	*-u*
"I"	Topic marker	"banana"	Accusative marker	"eat" (verb stem)	Verb inflection connecting verbs or auxes	Progressive aux	Politeness	Declarative

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

Sentence 2) shows its free word-order. Given the relatively free word order, sentence 2) is also grammatical, with the same meaning as sentence 1).

2) *banana-wo* *watashi-wa* *tabe-te-i-mas-u* .

Also it has many allomorphs. One of them presented in 3).

3) *tabemashita = tabe-mas-(i)ta*

Verb stem "eat" + Polite "mas" + Past "(i)ta"

➔ (I) ate.

Here, *-ta* and *-ita* are allomorphs representing past tense declarative. Then, *-ita* is epenthesized for phonological reasons influenced by preceding affricate sound *-s-*, becoming *-hita*.

Implementation

Our main goal was to implement agglutinativity of the Japanese language using the feature unification algorithm given in P.hs. 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. But before that, we started work on the parser by first making everything that occurred before the main verb work, since Verbs are a whole problem unto themselves.

Nouns

Lexicon

The easy part about Japanese is that nouns do not have agreement, except in one case. The main verb "to exist/to have" has two forms - *iru* and *aru*. *Iru* only accepts subjects that are animate. *Aru*, in the other hand, only accepts inanimate subjects. Therefore we only needed the features [Anim] or [Inanim] for nouns. We kept First, Second, Third, Masc, Fem features for pronouns, but these have no bearing on agreement so they could also be left out.

4)

kodomo	*-ga*	*i*	*-ru*
"Children"	Nominal case mark	Main verb "exist/have"	Casual Decl ending

➔ (I) have children. / There are children. / There is a child.

5)

Penn	*-ga*	*ar*	*-u*
"pen"	Nomial case mark	Main verb "exist/have"	Casual Decl ending

➔ (I) have a pen. / There is a pen.

In order to mark case on NPs in Japanese, case marking particles are used. These particles all have their own feature pertaining to the grammatical case they bestow upon a noun. If we had subcategorization (discussed below), these would be the primary way to step generation of ungrammatical complements.

Japanese syntactically does not have Determiners or Adjectives like in English, but they are often referred to this way. The three determiners we have included - *kono, sono, ano* are actually nouns, but we can use the DET rules that came with the original parser to model them, so we kept them as determiners.

There are two classes of adjectives in Japanese: Na-Adjectives[Nadj] and I-Adjectives[Iadj].
Na-Adjectives syntactically behave like a special class of nouns, so we are treating them as nouns.
I-Adjectives end in *"ii"* and syntactically behave like a special class of verbs, so we have modelled them like verbs in our lexicon.

Parser

Luckily, Japanese has no person, gender, or number agreement. So any determiner, adjective, and case-marker can occur together with almost any noun. So we modified the original NP rule (npRule) to no longer check for agreement. We then needed to be able to parse the case marking particles, so we wrote 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 our implementation. In this partpRule, it needs to parse and match a NP and then multiple particles, since Japanese allows for multiple particles on NPs:

6)

curcuru	*-sann*	*-ni*	*-mo*	*tegami wo ka-(i)te-kur-e*
Curcuru	Polite vocative case mark	Dative case mark	Postposition "also"	Imperative VP: Write a letter

➔ (Write a letter) to Mr. Curcuru as well.

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, using combine function, the feature unification in P.hs. 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 all dependents in a list (e.g. a list of particles), and then concatenating all those features into one list. Since combine performs agreement checks, we can prevent over-generating parses.
We also changed PPs from Prepositional Phrases to Postpositional phrases, with postpositions coming after the NP they govern.

S-level

Parser

Next we had to change the original sRule to account for free word order. We did this similarly to how the original vpRule's subcategorization 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:

7)

watashi	*no*	*tabe*	*-ta*	*banana*	*wa oishi-kat-ta des-u*
"I"	Genetive Mark	"eat"	Past	banana	was tasty

➔ The banana that I ate was tasty.

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.

Verb

Lexicon

Now for the hard part: the Verbs. Agglutination is awful. We created a complex system to model the agglutinative nature of verb morphology. We have four types of morphemes to complete a VP, and we labeled them Stem, END, INF, FIN. Every Japanese verb phrase needs at least a stem (real verb with its content) and one ending (grammatical function word, the most base case is declarative ending *'u'* which is unmarked at all in terms of tense, mode, honorifics, aspect, mood, and any other grammatical category). However, a speaker can arbitrarily add a number of semantic features in those categories to a verb. Thus between the stem and the sentence-finalizing ending (we tagged these "FIN" – finalizing ending) there can be a random number of other morphemes, and there are dependencies between them – some can go with some but not others. Also of note is that some of them are semantically meaningful (e.g. indicating tense or modality, etc), while others only have morpho-syntactic / phonological roles that connect two morphemes preceding and following them. We named the former "END" – endings, and the latter "INF" – inflections.

We tried to model this by having the features block unification of combinations that are not grammatical. In order to determine which morphemes can transition to other morphemes, a system of features is in place that blocks ungrammatical transitions.

In principle, Japanese verbs largely fall into two classes; a group of verbs whose stems end with consonants (godan verbs, [Dan5] in our model), and another group whose stems end with vowels (ichidan verbs, [Dan1] in our model. Other than those, like all other natural languages, there are irregular verbs. We skipped including irregulars only to simplify our model and for the sake of time. However their conjugations are very fixed, so it would not be a strenuous task to add them in the future.

Japanese has a bunch of grammatical modes and voices; indicative/declarative, interrogative[Intrg], interjectory[Intrj], imperative[Imper], volitional[Volit] (let's …), hypothetical[Hypo], causative[Caus], passive[Pass]. Among them we succeeded to model all but interrogative, imperative, and hypothetical moods. Progressive aspect, and modalities of potential & necessity are expressed with auxen in Japanese, which will be described below. Each grammatical ending chooses a specific form of its main verb, and in fact, some endings transform its host into a different type or form. For example, negative ending "nai" itself in the form of I-adjective, and when it attaches to a verb, the whole verb now behaves like a I-adjective. This lead us to enhance the given unification algorithm to a new level of feature replacement/update, which is also described in detail below. To build a robust model of

morpho-syntactic conjugations, we modelled all of inflectional forms (INF) of each kind of verb and tried to include at least one semantic ending (END) that chooses each INF. As a result we included 7 forms [Stem], [Nai], [Te], [Ta], [Masu], [Reru], [You]. Table 1 shows examples of ending that requires each form

Stem	"u" unmarked declarative finalizing ending
Nai	"na" negative ending
Te	"te" ending that compounds two verbs
Ta	"ta" past tense finalizing ending
Masu	"mas" polite declarative ending
Reru	"re" potential model ending
You	"you" volitional mood finalizing ending

Table 1) verb inflectional forms and their corresponding ending

Parser

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 cases 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. We have rewritten this rule many times (I stopped counting after 5). The original implementation was:

```
vpRule :: PARSER Cat Cat
vpRule = \xs ->
 [ (Branch (Cat "_" "V" fs []) (vp:xps),zs) |
   (vp,ys)  <- leafP "V" xs,
   (xps,zs) <- parseFins ys,
   fs       <- superCombine vp xps,
   and (map (\x -> agreeC vp x) xps)]

parseEndings :: PARSER Cat Cat
parseEndings = parseFin <|> parseEnd <|> parseInf

parseFins :: [Cat] -> [([ParseTree Cat Cat],[Cat])]
parseFins = many parseEndings
```

This rule parsed for a verb stem, and then looked for any combination of tokens with the INF, END, or FIN tags. It then used superCombine function to combine the features of the stem with all the features of the ending morphemes. It also checked if the verb stem agrees with every morpheme that was attached. This worked, and allowed for parsing verbs of arbitrary length that did not have conflicting features. But we had a problem: As verbs are built up from stem to ending, the intervening morphemes change what remaining morphemes can be attached. So we needed to simulate this building up and check that the features matched after each morpheme attached, not just all at once. So we changed two things - how our vpRule worked and how the "combine" function worked.
For the vpRule we simply turned it into 3 VP rules: one to combine V and INF, one for V and END, and one for V and FIN. The one for INF simply parsed a VP Stem before parsing for a INF. The one for END first parsed either a VP stem or a INF VP. The one for FIN first parsed either a VP stem, or a INF VP, or a END VP.

```
infVpRule :: PARSER Cat Cat
infVpRule = \ xs ->
   [ (Branch (Cat "_" "V" fs []) [vp,inf],zs) |
        (vp,ys)  <- leafP "V" xs,
        (inf,zs) <- parseInf ys,
        fs     <- combine (t2c vp) (t2c inf),
        agreeC vp inf ]

parseInfVP :: PARSER Cat Cat
parseInfVP = leafP "V" <|> infVpRule

endVpRule :: PARSER Cat Cat
endVpRule = \ xs ->
   [ (Branch (Cat "_" "V" fs []) [vp,end],zs) |
        (vp,ys)  <- parseInfVP xs,
        (end,zs) <- parseEnd ys,
        fs     <- combine (t2c vp) (t2c end),
        agreeC vp end ]

parseInfEndVP :: PARSER Cat Cat
parseInfEndVP = leafP "V" <|> endVpRule <|> infVpRule

finVpRule :: PARSER Cat Cat
finVpRule = \ xs ->
```

```
[ (Branch (Cat "_" "V" fs []) [vp,fin],zs) |
    (vp,ys)  <- parseInfEndVP xs,
    (fin,zs) <- parseFin ys,
    fs       <- combine (t2c vp) (t2c fin),
    agreeC vp fin ]
```

The given feature unification function combine basically takes two parse trees and merges all of their features. Then it checks if there are two or more features in a single category, which means unification has failed. For instance, a merged node can't have [Anim] and [Inanim] at the same time, since they are both in the Animacy category. As we took advantage of this function, our initial implementation just represents each form as a feature in the verbForm category. See sentence 8), 9) for examples:

8)

"ar-"/V[Stem] + "-u"/FIN[Stem] → *"aru"/VP*

"exist" declarative "there exists"

But,

9)

"ar-"/V[Stem] + "-nai"/END[Nai, Nega] → CLASH!

Then we found out that mere unification did not work for inflectional endings. In sentence 9) "nai"END should take a host in [Nai] form, and there is an inflectional ending "a"/INF that transform the host verb into [Nai] form. See sentence 10)

10)

"*ar*-"/V[Stem] + "-*a*-"/INF[Stem, Nai] → "*ara*-"/VBAR[Stem, Nai]

"*ara*-"/VBAR[Stem, Nai] + "-*nai*"/END[Nai,Nega] → "*aranai*"/VP[Stem, Nai, Nega]

➔ "There is no…"

In this example, the feature unification is not licit. After the first line, the host node bears two features in verbFormFs category. We need to 'update' the [Stem] feature of the verb stem to [Nai] when it is actually in [Nai] form after combining with "-a-". So we decided to enhance the combine function to perform not only feature unification, but also feature update. We prepared some special feature sets repType, and repForm, to notify the combine function that a type feature and/or form feature on the host's side needs to be update. And then it uses Haskell's built-in filter and map to update features. See 11) for an example of fully derived feature unification & update for "*ararani*" :

11)

"*ar*-"/V[Dan5, Stem] + "-*a*-"/INF[Stem, **Tfnai**] → "*ara*-"/VBAR[**Nai**]

"*ara*-"/VBAR[Dan5, Nai] + "-*na*-"/END[Nai, **Tyiadj**, **Tfstem**, Nega] → "*arana*-"/V [**Iadj**, **Stem**, Nega]

"*arana*-"/VBAR[Iadj, Stem, Nega] + "-*i*"/FIN[Iadj, Stem, Decl] → "*aranai"*/VP[Iadj, Nega]

Here are implementation of enhanced combine and its helper functions

```
transfrom :: Feat -> Feat
transfrom s
  | s == Tpdan5 = Dan5
  | s == Tpdan1 = Dan1
  | s == Tpiadj = Iadj
  | s == Tfstem = Stem
  | s == Tfnai  = Nai
  | s == Tfte   = Te
  | s == Tfta   = Ta
  | s == Tfmasu = Masu
  | s == Tfreru = Reru
  | s == Tfyou  = You
  | otherwise   = Wh

repFilter :: Cat -> Agreement -> [Agreement] -> [Agreement]
repFilter cat filt feats
  | filt == repFormFs && length (filter (`elem` filt) (fs cat)) >= 1
      = map (filter (not . (`elem` repFormFs)))
            (map (filter (not . (`elem` verbFormFs))) feats)
  | filt == repTypeFs && length (filter (`elem` filt) (fs cat)) >= 1
      = map (filter (not . (`elem` repTypeFs)))
            (map (filter (not . (`elem` verbTypeFs))) feats)
  | otherwise = feats

combine :: Cat -> Cat -> [Agreement]
combine cat1 cat2
  | length replace >= 1
      = map (++ (map transfrom replace))
        (repFilter cat2 repTypeFs
         (repFilter cat2 repFormFs
          (rawCombine (fs cat1 ++ fs cat2)) ) )
  | otherwise
      = rawCombine (fs cat1 ++ fs cat2)
 where replace = (repType (fs cat2) ++ repForm (fs cat2) )

rawCombine :: Agreement -> [Agreement]
rawCombine fss =
 [ feats | length (gender   feats) <= 1,
           length (person   feats) <= 1,
           length (gcase    feats) <= 1,
           length (pronType feats) <= 1,
```

```
          length (tense    feats) <= 1,
          length (postType feats) <= 1,
          length (verbType feats) <= 1,
          length (verbForm feats) <= 1,
          length (honorif  feats) <= 1,
          length (animacy  feats) <= 1]
  where
    feats = (nub . sort) fss
```

Because of all the headaches with VPs, we were not able to fully implement Auxen. Auxen in Japanese behave in a straightforward way, however. When a Verb ends in [Te] form, an Aux can be attached to the right hand side, just like any other verb morpheme. And because it is the start of a new verb, there are no morpheme agreement checks other than the check for [Te] form. Then an Aux behaves just like a main V: it has a stem, and can take all the same morphemes that a main V can. This means that it can also end in [Te] form, and so there can be a chain of Auxen attached to a main V.

There are lots of Auxen in Japanese, and each carry different meanings. However, they have a certain order they need to respect - some Auxen must precede other Auxen, just like how some verb morphemes must precede others. So a key to implementing them would be to figure out a way to restrict this relative ordering.

But other than that problem, implementing them should be relatively straightforward. You would just need a rule that would take a VP that ends in [Te] form and an AuxVP and makes a VP out of them. Then you might want to have some semantic feature for each Aux that could be added to the overall feature set for the VP. And then when adding a new Aux, you would need to check if there's a feature in the VP set that would bar this new Aux from being added (i.e. to make sure the Auxen ordering is upheld).

Currently our parser is able to attach an Aux to a [Te] form. You can even add another Aux if the first Aux ends in [Te] form. However the running time seems to slow down considerably, and gets even worse if you try adding another Aux.

Some examples to show how our parser works:

Japanese has no white-spacing word separation. Word segmentation itself is an intricate NLP task, thus here we take segmentation as given. Each white space in the below examples means morpheme-level segmentation.

Free word order, main verb in Te form with an added Aux *i-ru* (progressive meaning):

prs "sensei wa jibun de eki kara yama made hito wo koros i te i ru"
The teacher herself from the station to the mountain is killing people.

prs "sensei wa banana wo eki kara yama made jibun de tabe te i ta"
The teacher herself from the station to the mountain was eating a banana.

prs "banana wo tabe ta"
(I) ate a banana.

Faked relative clauses, here just parses as PartP VP PartP VP:
prs "watashi no tabe ta banana ga shin nn ta"
The banana that I ate died.

prs "kuma wa yama ga tabe rare ru"
The bear can eat the mountain.

Noun + copula:
prs "sono jitensha wa asoko desu"
That bike is over there.

prs "tabe you"
Let's eat!

Not all grammatical combinations of stem + morphemes are currently working, for example:

prs "tabe mas u"

I eat. (polite)