Supplementary A. Built-in transition systems

In the following, we document all transition systems built in PanParser, along with their cost. s and s' denote top elements of the stack σ, b denotes the head of buffer β, and P is the set of edges representing the partially built tree. The tokens that are not words are the padding tokens (<start> and ROOT). For the action costs, we also provide compact representations of each setting yielding non-zero costs, using σ to represent 'any stack element' (and β for 'any buffer element') and h^*_w is the reference head of word w.

The partial and short-spanned systems are new transition systems designed as part of PanParser. All others are drawn from the literature, but in general their properties had not yet been studied for both ROOT positions.

A.1. ArcEager

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \to s))$	if s is a word and s is unattached
RIGHT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s\|b,$	$\beta,$	$P + (s \to b))$	
REDUCE	$(\sigma\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$\beta,$	$P)$	if s is attached

Table 1: Action semantics for the ARCEAGER transition system.

Case with ROOT in first position When the ROOT token is in first position, the upper description of the ARCEAGER system (Nivre, 2004) requires some amendments.[1]

First, if affects the preconditions of some actions. Indeed, SHIFT becomes illegal when the buffer contains a single element, because afterwards it cannot be attached, and the final configuration with ROOT token requires an empty stack. Similarly, RIGHT is illegal when the buffer contains a single element and at least one word in the stack is unattached.

Second, as shown in Figure 1, it makes the system non-arc-decomposable, as Goldberg and Nivre (2013) does not hold anymore. There is exactly one configuration which embeds arc incompatibilities. It is the case when the reference ascendance of the last word in sentence (n_N) consists in any number of buffer tokens followed by a stack element, and deeper in the stack (including n_N's ancestor) at least one word is still unattached. Because of the extra preconditions, in that case at least one ancestor of n_N will not get its correct head, but will instead be promoted as head of the unattached stack element. Figure 2 illustrates this configuration.

[1] To the best of our knowledge, this topic is not addressed in the literature.

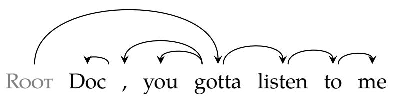
SHIFT	$(\sigma, \ b\vert\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack
	$(\sigma, \ b\vert\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT	$(\sigma\vert s, b\vert\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\vert s, \ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT	$(\sigma, \ b\vert\beta)$	$b \curvearrowleft \beta$	b if h^*_b is in buffer but not on top
	$(\sigma\vert s, b\vert\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma, \ b\vert\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
REDUCE	$(\sigma\vert s, \ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

(a) Tokens that explicitly lose their head.

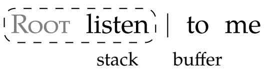
SHIFT	$(\sigma\vert s, \ b\vert\beta)$	$? \curvearrowright b$	b if h^*_b is on the left and the stack is not empty
LEFT	$(\perp\vert s, \ b\vert\beta)$	$? \curvearrowright b$	b if h^*_b is on the left and the stack contains a single token
RIGHT	$(\sigma, b\vert b'\vert\beta)$	$b \curvearrowleft ?$	b if h^*_b is on the right and the buffer contains at least two tokens
REDUCE	$(\perp\vert s, \ b\vert\beta)$	$? \curvearrowright b$	b if h^*_b is on the left and the stack contains a single token

(b) Tokens whose specified head direction is explicitly forbidden.

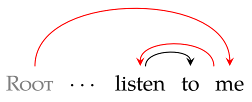
Table 2: Action cost of the ARCEAGER transition system.



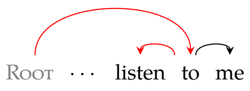
(a) Reference parse tree.



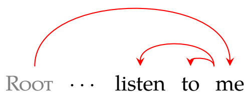
(b) Stack and buffer of the (already suboptimal) configuration to evaluate; 'listen' is unattached. Possible actions are RIGHT, LEFT and SHIFT.



(c) Case RIGHT: best parse with REDUCE-LEFT-RIGHT-REDUCE afterwards.



(d) Case LEFT: best parse with RIGHT-RIGHT-REDUCE-REDUCE afterwards.



(e) Case SHIFT: afterwards LEFT-LEFT-RIGHT-REDUCE is enforced.

Figure 1: Counter-example to arc-decomposability of the ARCEAGER transition system with ROOT in first position.

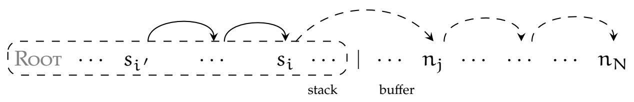


Figure 2: Prototype of arc incompatibilities for ARCEAGER with ROOT in first position.

Consequently, non-arc-decomposability adds a (relaxed) cost of 1 if the parser is in this configuration, or will be put in this configuration by the given action. More precisely, a single arc incompatibility is inserted the first time that an ancestor of the last token is shifted, or attached to a stack containing at least one unattached word.

Additional cost: head direction constraints As an experimental feature, the ARCEAGER system provides the possibility to compute the action cost when only the dependency direction is known. This mostly allows to parse under simpler constraints, for instance built using typological knowledge. Table 2b documents the additional cost incurred by such annotations.

A.2. ArcHybrid

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \rightarrow s))$	if s is a word
RIGHT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \rightarrow s))$	

Table 3: Action semantics for the ARCHYBRID transition system.

The ARCHYBRID system has been proposed by Kuhlmann et al. (2011) as a compromise between ARCSTANDARD and ARCEAGER properties.

In this system, changing the ROOT position has no effect on preconditions, since the top of the stack can always be attached, either on the left or on the right. The cost is also unchanged, as the ARCHYBRID system is always arc-decomposable.

A.3. ArcStandard

For the ARCSTANDARD system, originally designed by Nivre (2003), note that there is no precondition on SHIFT. Indeed, when the ROOT token is in last position, it must be shifted to receive its children and reach a final configuration. The dynamic oracle for this system directly computes COST as a loss difference, following the methodology proposed by Goldberg et al. (2014).

SHIFT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT	$(\sigma\|s,\ \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 4: Action cost of the ARCHYBRID transition system: tokens that explicitly lose their head.

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	
LEFT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s,$	$\beta,$	$P + (s \rightarrow s'))$	if s' is a word
RIGHT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \rightarrow s))$	

Table 5: Action semantics for the ARCSTANDARD transition system.

A.4. NonMonotonicArcEager

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \rightarrow s))$	if s is a word
RIGHT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s\|b,$	$\beta,$	$P + \mathit{Temporary}(s \rightarrow b))$	if b is a word
REDUCE	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \rightarrow s))$	

Table 6: Action semantics for the NONMONOTONICARCEAGER transition system.

The NONMONOTONICARCEAGER system (Honnibal et al., 2013) is in fact very similar to ARCHYBRID, with the RIGHT transition renamed as REDUCE and a second shift transition, called RIGHT. However, even if both have the same expressivity and search space, at feature level the NONMONOTONICARCEAGER system allows to enrich the representation with knowledge of highly probable heads, which may help some decisions.

Still, this does not affect the action cost, which is the same as for ARCHYBRID. Table 7b provides additional costs for head direction constraints, similarly to the ARCEAGER case.

SHIFT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
REDUCE	$(\sigma\|s,\ \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

(a) Tokens that explicitly lose their head.

LEFT	$(\perp\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h^*_b is on the left and the stack contains a single token
	$(\sigma\|s'\|s,\ \beta)$	$? \curvearrowright s$	s if h^*_s is on the left and the stack contains at least two tokens
REDUCE	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft ?$	s if h^*_s is on the right and the buffer is not empty
	$(\perp\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h^*_b is on the left and the stack contains a single token

(b) Tokens whose specified head direction is explicitly forbidden.

Table 7: Action cost of the NONMONOTONICARCEAGER transition system.

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$		if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \rightarrow s))$		if s is a word and s is unattached
RIGHT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s\|b,$	$\beta,$	$P + (s \rightarrow b))$		
REDUCE	$(\sigma\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$\beta,$	$P)$		if s is a word

Table 8: Action semantics for the ARCEAGERPARTIAL transition system.

A.5. ArcEagerPartial

ARCEAGERPARTIAL is our proposal for an ARCEAGER parser that learns to *reproduce* partial annotations, i.e. it both learns syntactic knowledge based on the provided annotations, and learns which tokens should remain unannotated because of a lack of information. This is close to confidence-based learning, except that the confidence criterion is embedded in the training data.

In practice, the ARCEAGERPARTIAL system extends ARCEAGER with the possibility to predict empty attachments as part of the system, and the final output is consequently a partial tree by design.

Empty attachments are encoded by SHIFT+REDUCE: the word is put on the stack as usual, can receive children, but then is popped before receiving its own head. Conse-

SHIFT	$(\sigma, \quad b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack
	$(\sigma, \quad b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT		$✗ \curvearrowright s \curvearrowleft ✗$	s if no h^*_s
	$(\sigma\|s, b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s, \quad \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT		$✗ \curvearrowright b \curvearrowleft ✗$	b if no h^*_b
	$(\sigma, \quad b\|\beta)$	$b \curvearrowleft \beta$	b if h^*_b is in buffer but not on top
	$(\sigma\|s, b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma, \quad b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
REDUCE	$(\sigma\|s, \quad \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer and s is unattached
	$(\sigma\|s, \quad \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 9: Action cost of the ARCEAGERPARTIAL transition system: tokens that explicitly lose their head.

quently, the only modification to the system semantics is to relax the precondition on REDUCE actions, and allow them even on unattached words.

The impact is stronger on the action cost. Compared to ARCEAGER, it indeed adds two cases of non-zero cost: when a word whose reference attachment is empty receives a head (to enforce reproduction of empty attachments), and when an unattached word is reduced while its reference head is still somewhere in the buffer (which cannot happen with ARCEAGER).

Note that when the dynamic oracle is used to filter actions based on partial constraints, the cost incurred by empty attachments is ignored, otherwise the partial tree could never be completed.

Finally, regarding the ROOT position, it does not affect ARCEAGERPARTIAL as it does for ARCEAGER, because with the relaxed precondition, it is always possible to reduce stack elements to reach a final configuration. The system is consequently always arc-decomposable.

A.6. ArcHybridPartial

ARCHYBRIDPARTIAL applies the same ideas of partial parsing, but on the ARCHYBRID system. In this case, there are only three actions, but it is in fact not possible to encode empty attachments with three actions. So, we *extend* the transition set with an additional REDUCE action, which operates as in ARCEAGERPARTIAL: it pops the first stack element, even if it is unattached (which is necessarily the case here).

Shift	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$		if b is a word
Left	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \to s))$		if s is a word
Right	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \to s))$		
Reduce	$(\sigma\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$\beta,$	$P)$		if s is a word

Table 10: Action semantics for the ArcHybridPartial transition system.

Shift	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
Left		$✗ \curvearrowright s \curvearrowleft ✗$	s if no h^*_s
	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
Right		$✗ \curvearrowright s \curvearrowleft ✗$	s if no h^*_s
	$(\sigma\|s,\ \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
Reduce	$(\sigma\|s,\ \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 11: Action cost of the ArcHybridPartial transition system: tokens that explicitly lose their head.

The resulting system consists in one shift action and three reduce actions, each one encoding a different attachment of s (s′, b and empty). Compared to ArcEagerPartial, the semantics of each action are better singled out.

For the basic actions, the cost of ArcHybridPartial is the same as ArcHybrid, augmented with enforcement of empty attachments (again, not in case of constraint-based filtering). The cost of Reduce is the same as the basic Right, except that it also penalizes reducing tokens whose head is on the left (i.e. when a Right is required).

A.7. Short-spanned dependencies

Finally, we also propose transition systems that only learn and predict the shortest dependencies. These systems, ArcEagerSpan and ArcHybridSpan, are based on the partial systems ArcEagerPartial and ArcHybridPartial, and parameterized by a maximum dependency length (at parser initialization time). For instance, if the maximum length is 1, the parser will only predict attachments to neighbouring tokens, leaving unattached the tokens that typically have long distance attachments in the training data.

The constraint on dependency length only affects the action preconditions by forbidding LEFT and RIGHT actions which would create too long dependencies. Otherwise, it is completely identical to the partial systems. The cost is also the same, because long dependencies are filtered out from the reference as a preprocessing step, thus resulting in partial annotations containing only short dependencies.

Supplementary B. Code examples

B.1. Parser usage

To train and test a new parser:

```
# load a CoNLL-X or CoNLL-U treebank
from dependency_parser import read_conll
with open("en-train.conllu", "rt") as fh:
    dataset = read_conll(fh)
# train a new parser with default parameters
from dependency_parser import train_dependency_parser
from misc.io import FrogressListener # for better display (optional)
parser = train_dependency_parser(None, dataset, n_epoch, listener=FrogressListener)
# annotate an input dataset and dump it in a CoNLL-U file
from dependency_parser import process_dependency_parser
from conll.io import wt, feat2col
with open("out.conllu", "wt") as out:
    wt(out, process_dependency_parser(parser, newdataset), preprocess=feat2col)
# print the parser UAS
from dependency_parser import eval_dependency_parser
print(eval_dependency_parser((parser, testset)))
```

The default is an (unlabeled) ArcEager perceptron parser with a beam of size 8, the Root token in last position, Zhang and Nivre (2011)'s feature templates (augmented with transition history), a global dynamic oracle and the early-update strategy.

To change these parameters:

```
parser = train_dependency_parser(dict(system="NonMonotonicArcEagerLabeled", root_first=True,
            beam=16, delexicalized=True), dataset, n_epoch, strategy="maxv")
# equivalent to
from functools import partial
from dparser.features import zn11_features
delex_fts = partial(zn11_features, delexicalized=True)
parser = NonMonotonicArcEagerLabeledBeamParser(delex_fts).with_beam(16).with_root_first(True)
train_dependency_parser(parser, dataset, n_epoch, strategy="maxv")
```

To use the greedy-specific implementation:

```
from dparser.features import zn11_features
parser = train_dependency_parser(dict(beam=None), dataset, n_epoch)
# equivalent to
parser = ArcEagerLocalParser(zn11_features)
train_dependency_parser(parser, dataset, n_epoch)
```

To define a custom transition system:

```
import dependency_parser, custom_system
from dparser.perceptron_parser import PerceptronParser
class CustomBeamParser(custom_system.BeamParser, PerceptronParser): pass
dependency_parser.CustomBeamParser = CustomBeamParser # add to built-in scope
parser = train_dependency_parser(dict(system="Custom", foo=bar), dataset, n_epoch)
```

To define a custom oracle/update strategy:

```
import dependency_parser as dp
def strategies(parser, strategy, builtin=dp.strategies):
    (time_limit, check_cut_edge, check_single_reference, oracle, restart,
     exploration_rate, beam_update, remember_bad_beams) = builtin(parser, strategy)
    if strategy == "customstrategy":
        oracle, time_limit, restart = EARLY_LONGEST_PREFIX, 4, True
    return (time_limit, check_cut_edge, check_single_reference, oracle, restart,
            exploration_rate, beam_update, remember_bad_beams)
def exploration_strategy(strategy, itn, n_epoch, builtin=dp.exploration_strategy):
    exploration_rate = builtin(strategy, itn, n_epoch)
    if strategy == "customstrategy":
        exploration_rate = 3*itn/epoch
    return exploration_rate
# replace built-in
dp.strategies = strategies
dp.exploration_strategy = exploration_strategy

parser = train_dependency_parser(None, dataset, n_epoch, strategy="customstrategy")
```

To extend the binary feature representations with pre-trained embeddings:

```
# extend binary features with embeddings
embeddings = {"<UNK>": [0,0.9,0.5,1], "DeLorean": [1,0,0.1,0],
              "Chicken": [0.8,0.1,0.3,1]}
custom_features = partial(zn11_features, embeddings=embeddings)
```

To have three parsers voting on each decision (with ties broken by `parser1`), first train the parsers separately, and then do:

```
from classifier.vote import VoteClassifier
parser1.model = VoteClassifier((parser1.model, 1.1), (parser2.model, 1), (parser3.model, 1))
# now parser1 acts according to the vote of all three parsers
print(eval_dependency_parser((parser1, testset)))
```

B.2. Tagger usage

```
from dependency_parser import read_conll
with open("en-train.conllu", "rt") as fh:
    dataset = read_conll(fh)
with open("en-test.conllu", "rt") as fh:
    testset = read_conll(fh)

# train and evaluate a perceptron tagger
from tagger.postagger import train_pos_tagger, eval_pos_tagger
from tagger.perceptron_postagger import PerceptronPOSTagger
from conll.io import feat2tag # to retain only coarse PoS tags
tagger = train_pos_tagger(PerceptronPOSTagger(), feat2tag(dataset), n_epoch)
print(eval_pos_tagger((tagger, feat2tag(testset))))
```

```
# tagging and parsing pipeline
from tagger.neural_postagger import FeedForwardPOSTagger
tagger = train_pos_tagger(FeedForwardPOSTagger(), feat2tag(dataset), n_epoch_tag)
# to train on predicted tags
from tagger.postagger import process_pos_tagger
dataset = process_pos_tagger(tagger, dataset)
parser = train_dependency_parser(None, dataset, n_epoch_parse)
# annotate new data
with open("en.conllu", "rt") as fh:
    newdataset = read_conll(fh)
from conll.io import wt, tag2col
with open("out.conllu", "wt") as out:
    wt(out, process_pos_tagger(tagger, newdataset), preprocess=tag2col)
```

B.3. Built-in utilities for error analysis

```
# ignore PUNCT (default), INTJ, SYM and X tokens in evaluation
eval_dependency_parser((parser, testset), ignore_tags=["PUNCT", "INTJ", "SYM", "X"])

# fine-grained scores depending on the (reference) in-tree depth of the tokens,
# capped to 4
eval_dependency_parser((parser, testset), by_type=MDEPTH)

# fine-grained scores for PoS tag pairs, e.g. attachment score over the nouns whose
# head is a verb
from misc.xp import uas_bitable
uas_bitable(eval_dependency_parser((parser, testset), by_type=(POS, HEADPOS)))
# the same, but only for tokens whose head is on the right
uas_bitable(eval_dependency_parser((parser, testset), by_type=(POS, HEADPOS),
                                   subset=(DIRECTION, "+")))

# fine-grained scores for all PoS-disambiguated words with "mark" attachment, i.e.
# keys are "except-ADP", "except-SCONJ", "like-ADP", "like-SCONJ", "with-ADP", etc.
from misc.xp import scores
scores(eval_dependency_parser((parser, testset), by_type=[WORD, "-", POS],
                              subset=(LABEL, "mark")))

# compute corpus statistics both on the direction-disambiguated head PoS for each
# child PoS ("2pos+dir" key), and on the dependency length ("L" key)
eval_dependency_parser(testset, by_type={"L": LENGTH,
                       "2pos+dir": (POS, [HEADPOS, DIRECTION])}, out=STATS)

# confusion matrix on the head PoS
from misc.xp import confusion_matrix
confusion_matrix(eval_dependency_parser((parser, testset), by_type=HEADPOS, out=CONFUSION))

# attachment agreement of two parsers, for each gold label
eval_dependency_parser([(p1, testset), (p2, testset)], by_type=LABEL, out=COMPARE)[2]
```

Supplementary C. Code-level architecture

Since all components of a parser continuously interact during training and prediction, the desired modularity is not natively achievable when implementing a parser; in PanParser, it is ensured by additional layers of abstraction, together with specific implementation choices. We briefly describe our approach in the following, providing technical details on three key aspects: the core framework that lets all components interact (§C.1), its extension to support beam search and global training (§C.2), as well as how the class hierarchy is leveraged to enable algorithmic combinations (§C.3).

C.1. Structured prediction framework

The whole framework revolves around the `AbstractStructuredPredictor` class, from which all taggers and parsers inherit. Its design is based on the idea of coroutines: the components interact via Python generators, each representing a different kind of data (prediction configurations, update configurations, predicted classes, output annotations, etc., as well as batched versions of all). This is what allows the classifier to be considered as a black box: it simply feeds from a stream of configurations, and provides a stream of predictions. The role of the `AbstractStructuredPredictor` class is to plug these generators together, and to add a feedback loop so that a given prediction can be both used as an output, and reused (by the task-specific component) to compute the next configuration.

Task-specific components Designing a task-specific component consists in implementing two methods. The `xs` method returns (given an input sentence) a function mapping a stream of classifier predictions to a (multiplexed) stream of feature vectors to classify and of output predictions (e.g. a tag or the k-best parse trees). `xys` does the same for training, i.e. yielding feature vectors to classify, as well as update pairs. Additionally, `polish` can be overridden: it does nothing by default, but can add post-processing of the final output, for instance gathering all tags into a list or selecting the top hypothesis of a k-best list.

The base class has another method to implement, `_defaultmodel`, but this is done when plugging the wanted classifier: it builds and returns a new classifier, initialized with the dataset.

Framework API A system built with this framework can then be used with only two methods, `train` and `process`. The former takes care of epochs, sampling, cross-validation and post-training refinements like averaging. The latter initiates dataset annotation, with pre- and post-processing to fill the appropriate slots with the tags, heads or labels predictions.

System customization There are two ways to add optional behaviors or parameters. The first is the `strategy`, `epoch_strategy` and `sample_strategy` methods: based on a strategy identifier and knowledge of the epoch, they return additional training parameters that `AbstractStructuredPredictor` ultimately feeds to `xys`. By default they are placeholders, overwritten by custom functions in `train_dependency_parser` and `train_pos_tagger`.

The second is a series of hooks, gathered in a `Callback` object sent to the `train`, `xys` and `xs` methods. They are called in various crucial places (before and after epochs, sample processing, updates, evaluation, etc.) and can be used for additional monitoring, or for more evolved purposes like on-the-fly subsampling.

Evaluation utilities The `AbstractStructuredPredictor` class also provides several basic utilities for evaluation and error analysis. The `eval_routine` function collects fine-grained accuracies for arbitrary category criteria, and `matrix_routine` builds a confusion matrix with such arbitrary groupings. `compare_routine` and `matrix_compare_routine` compare these measures between two systems. Finally, `stats_routine` provides various statistics on the dataset itself, using the same kind of criteria. All these functions operate on annotated datasets; testset annotation is rather handled at the level of `eval_dependency_parser` and `eval_pos_tagger`.

C.2. Beam search and global training strategies

Beam parsers are also based on the structured prediction framework, but they require extra generic functionalities, which the `AbstractBeamStructuredPredictor` class provides.

The `lookahead` method is in charge of the actual beam search. Given a stream of classifier predictions and an initial state for the beam (in most cases, a single configuration), it repeatedly extends the beam until an erroneous state is found, then yields all subsequent states, until final configurations. It consequently returns a multiplexed generator of feature vectors to classify, and erroneous states (i.e. k-best hypotheses). Using generators makes decoding lazy, so that in case of early update it can stop after the first error and no extra computation is done. At each extension of the beam, `lookahead` also enriches the hypotheses with aggregated information on the action costs, so that erroneous states are detected in constant time. Note that the gold actions are those with minimum cost and not zero cost: Appendix A of the paper explains why.

The `_infer` method is in charge of searching for update configuration pairs. Given a stream of classifier predictions and a sentence, it repeatedly calls `lookahead` (initially, on an empty configuration), processes the erroneous states to find an update pair, then selects a new initial configuration and goes on, until the sentence is processed. Here again, restart is lazy. To select update configurations, `_infer` applies the specified training strategy, thanks to three oracle utilities: `forced_decoding` to perform a lookahead with gold actions only, `lookfurther` to search for a max-violation state among those returned by `lookahead`, and `refchoice` to select a positive configuration when there are several.

When no error criterion is given (at prediction time), by design `lookahead` and `_infer` perform a single decoding pass, until full processing.

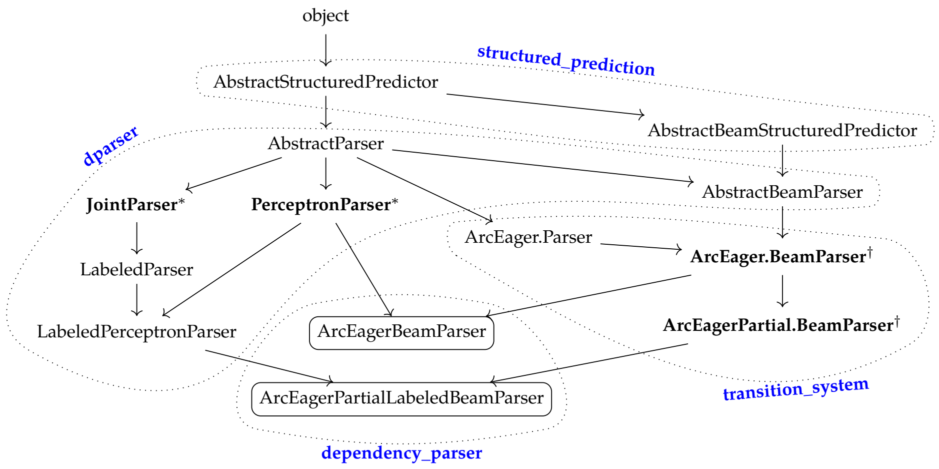
In the `AbstractBeamParser` class, the `search` and `learn` methods (returned respectively by `xs` and `xys`) both use `_infer`. `search` gets a beam of final hypotheses and extracts the k-best parse trees from these derivations. `learn`, for each update pair, goes through their history to extract feature vectors for classifier updates.

C.3. Class hierarchy

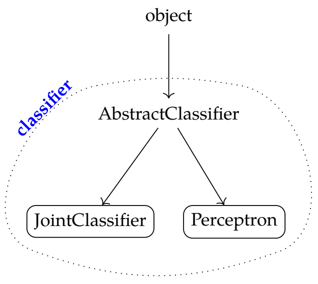
Because it focuses on modularity and maximal code factorization, PanParser contains a lot of Python classes, each bringing an additional piece of parser design.

For instance, the `ArcEager.Parser` class registers the set of transitions into the parser, forwarded to the classifier as possible classes by `PerceptronParser`, `ArcEager.TransitionSystem` defines the legal transitions in a given configuration and their action cost, while `ArcEager.BeamParse` provides their semantics, i.e. the effect of each action on the parse configuration, and how to output a parse tree based on the derivation.

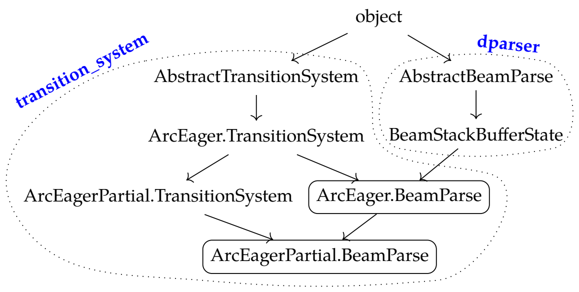
For this reason, and as a reference, we picture in Figure 3 the full class hierarchy of two parser classes: `ArcEagerBeamParser` for a standard case, and `ArcEagerPartialLabeledBeamParser` for a particularly complex case.



(a) Parser classes.



(b) Classifier classes.



(c) Parse configuration classes.

Figure 3: Class hierarchy for the `ArcEagerPartialLabeledBeamParser` and `ArcEager-BeamParser` classes, which are those actually used to build and train parsers. Classes denoted in bold with * contain references to classifier classes (`JointClassifier` and `Perceptron`). Classes denoted in bold with † contain references to parse configuration classes (`BeamParse` classes). Package names are indicated in blue.

Bibliography

Goldberg, Yoav and Joakim Nivre. Training Deterministic Parsers with Non-Deterministic Oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414, 2013. ISSN 2307-387X. URL `https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/145`.

Goldberg, Yoav, Francesco Sartorio, and Giorgio Satta. A Tabular Method for Dynamic Oracles in Transition-Based Parsing. *Transactions of the Association for Computational Linguistics*, 2: 119–130, 2014. ISSN 2307-387X. URL `https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/302`.

Honnibal, Matthew, Yoav Goldberg, and Mark Johnson. A Non-Monotonic Arc-Eager Transition System for Dependency Parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172, Sofia, Bulgaria, 8 2013. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W13-3518`.

Kuhlmann, Marco, Carlos Gómez-Rodríguez, and Giorgio Satta. Dynamic Programming Algorithms for Transition-Based Dependency Parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, Oregon, USA, 6 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-1068`.

Nivre, Joakim. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*, IWPT 2003, Nancy, France, 2003.

Nivre, Joakim. Incrementality in Deterministic Dependency Parsing. In Keller, Frank, Stephen Clark, Matthew Crocker, and Mark Steedman, editors, *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain, 7 2004. Association for Computational Linguistics.

Zhang, Yue and Joakim Nivre. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA, 6 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-2033`.