# Developing a Minimalist Parser for Free Word Order Languages with Discontinuous Constituency

Asad B. Sayeed and Stan Szpakowicz

School of Information Technology and Engineering
University of Ottawa, 800 King Edward Avenue
Ottawa, Ontario, Canada K1N 6N5
{asayeed@mbl.ca,szpak@site.uottawa.ca}

**Abstract.** We propose a parser based on ideas from the Minimalist Programme. The parser supports free word order languages and simulates a human listener who necessarily begins sentence analysis before all the words in the sentence have become available. We first sketch the problems that free word order languages pose. Next we discuss an existing framework for minimalist parsing, and show how it is difficult to make it work for free word order languages and simulate realistic syntactic conditions. We briefly describe a formalism and a parsing algorithm that elegantly overcome these difficulties, and we illustrate them with detailed examples from Latin, a language whose word order freedom causes it to exhibit seemingly difficult discontinuous noun phrase situations.

## 1 Introduction

The Minimalist Programme as described by Chomsky and others [Uriagereka, 1998] seeks to provide an explanation for the existence of the capacity of language in humans. Syntax merits particular attention in this programme, as it is syntax that mediates the interactions between the requirements of articulation and those of meaning. Minimalism characterizes syntax as the simplest possible mapping between semantics and phonology. It seeks to define the terms of simplicity and determine the structures and processes required to satisfy these terms.

That the object of investigation is syntax suggests that it is possible to extract a formal and computable model of syntactic processes from minimalist investigations. Doubly so, as the concept of economy in minimalism can be said to correspond to computational complexity.

In this paper, we look at the problem of developing a minimalist account of parsing. We turn our attention in particular to free word order phenomena in Latin and to simulating a realistic human parser given that people do not have a complete sentence before they begin processing. We first give background on free word order phenomena and on minimalist parsing. Next we discuss a formalism that performs free word order parsing in such a realistic manner. We show two complete parses of representative sentences to demonstrate the algorithm, one

of which is a case of discontinuous constituency, a special challenge for parsing languages such as Latin.

## 2 Background

### 2.1 Latin and Free Word Order

Principle-based parsing of free word order languages has been considered for a long time—see, for example, Kashket [1991]—but not much in the context of the Minimalist Programme. We propose a minimalist parser and illustrate its operation with Latin, a language that exhibits high word order freedom. For example,

| | | | | |
|---|---|---|---|---|
| pater | laetus | amat | filium | laetum |
| father-Nom | happy-Nom | loves-3Sg | son-Acc | happy-Acc |

'The happy father loves the happy son.'

For a simple sentence such as this, in theory all $5! = 120$ permutations should be grammatical. We briefly discuss in section 4 whether this is really the case. This is not true of Latin sentences in which function words must be fixed in place.

It is often remarked that these word orders, though semantically equivalent, differ in pragmatic import (focus, topic, emphasis, and so on). Existing elaborate accounts of contextual effects on Latin word order [Pinkster, 1990] are rarely sufficiently formal for parser development, and do not help a parser designed to extract information from a single sentence out of context. It should still be possible to extract the propositional content without having to refer to the context; hence, we need an algorithm that will parse all 120 orders as though there were no real differences between any of them. After all, people can extract information from sentences out of context.

### 2.2 Derivational Minimalism

Stabler [1997] defines a minimalist grammar as $G = (V, Cat, Lex, F)$. $V$ is a set of "non-syntactic features" (phonetic and semantic representations), $Cat$ is a set of "syntactic features," $Lex$ is the lexicon ("a set of expressions built from $V$ and $Cat$"), and $F$ is "a set of partial functions from tuples of expressions to expressions"—that is, structure-building operations such as MOVE and MERGE. MERGE, a binary operation, composes words and trees into trees. MOVE removes and reattaches subtrees; these manipulations are performed in order to *check* features. Checking ensures, among other things, that words receive required complements. Stabler characterizes checking as the cancellation of corresponding syntactic features on the participant lexical items, but requires that features be checked in a fixed order.

Stabler [2001] proposes a minimalist recognizer that uses a CKY-like algorithm to determine membership in $L(G)$. Limiting access to features to a particular order may not work for free word order languages, where words can often

appear in any order. Either duplicate lexical entires must proliferate to handle all cases, or Stabler's specification requirements of lexical entries must be relaxed. We choose the latter path.

Stabler's CKY inference rules do not themselves directly specify the order in which items are to be MERGEd and MOVEd into tree structures. This apparent nondeterminism is less significant if the feature order is fixed. Since we propose a relaxation of the ordering constraint for free word order languages, the nondeterminism will necessarily be amplified.

This dovetails with a practical goal. In free word order languages, semantically connected words can appear far apart in a sentence. In *pater amat filium laetum laetus*, *pater* and *laetus* must be merged at some point in a noun-adjective relationship. But *pater* is the subject of *amat*. In order to simulate a human listener, the parser would have to MERGE *pater* and *amat* first, despite that *pater* and *laetus* form one NP; the human listener would hear and make the connection between *pater* and *amat* first.

Consequently, we propose a parser that simulates a human listener by limiting at each step in a derivation what words are accessible to ("have been heard by") the parser and by defining the precedence of the operators in a way that fully extracts the syntactic and semantic content of the known words before receiving further words. We develop a formalism to allow this while providing the flexibility needed for free word order languages. This formalism, illustrated later by examples, reflects many of the ideas of Stabler's formalism, but is otherwise independent.

## 3 The Parser

### 3.1 Lexicalized Grammatical Formalism

We briefly describe enough of the lexicon structure to assist in understanding the subsequent parsing examples. A lexical entry has the form

$$\alpha : \Gamma$$

$\alpha$ is a word's phonetic or orthographic representation as required. $\Gamma$ is a set of feature structures, henceforth called "feature sets."[1] $\Gamma$ contains feature paths, described below. But more fundamental are the feature bundles from which feature paths are constructed.

Feature bundles are required given the fact that highly inflected languages often compress several features into a single morpheme. Feature bundles provide

[1] There is a third, hidden entity here: the semantic representation of $\alpha$. We leave it implied that parsing operations perform semantic composition; the formal specification of this is left for future work, but it can be specified in the lambda calculus as in Niyogi [2001] or via theta roles, and so on. Nevertheless, this paper is ultimately directed towards laying the syntactic groundwork for the extraction of semantic content from free word order sentences.

the means to check them simultaneously. A feature bundle is represented as follows:

$$\beta(\tau_1 : \phi_1, \ldots, \tau_n : \phi_n) : \delta, n \geq 1$$

$\beta$ is the feature checking status. It can be `unchecked` (`unch`), `UNCHECKED` (`UNCH`), `checked` (`ch`), `CHECKED` (`CH`), `unchecked-adjoin` (`unch+`), `checked-adjoin` (`ch+`). When feature bundles are checked against one another, $\beta$ can change. The examples will illustrate the relation between feature checking status symbols during the checking.

Each $\tau$ is a feature type. It can be one of such things as case, gender, number, and so on. Each $\phi$ is a feature value such as Nom (nominative), Sg (singular), and so on. The correspondence between feature types and features is required for the unification aspect of the checking operation, again demonstrated in the examples.

$\delta$ is direction. An item $\iota$ carrying the feature bundle can only check the bundle with a bundle on another item to the $\delta$ of $\iota$. $\delta$ can be *left* or *right*, and $\delta$ is omitted when the direction does not matter (a frequent situation in free word order languages).

A feature path has the form:

$$\pi \rightarrow \Gamma$$

$\pi$ is a feature bundle and $\Gamma$ is a feature set. A feature path makes $\Gamma$ inaccessible for checking until $\pi$ has been checked. $\Gamma$ can be empty, in which case the $\rightarrow$ is not written.

A feature set is simply an unordered list of unique feature paths: $\{\eta, \ldots\}$. These sets allow each $\eta$ to be checked independently of the others. In our representation of lexicon entries, we leave out the braces if there is only one $\eta$.

### 3.2 Data Structures and Operations

A parse (also known as a derivation) consists of a number of steps. Each step is of the form:

$$\Phi \mid \Psi$$

$\Psi$ is the queue of incoming words. It is used to simulate the speaker of a sentence. The removal of words is restricted to the left end of the queue. A word is shifted onto the right end of $\Phi$, given conditions described below. Shifting is equivalent to "hearing" or "reading" the next word of a sentence. $\Phi$ is a list, the processing buffer. It is a list of trees whereon the parser's operations are performed. The initial state of a parse has an empty $\Phi$, and the final successful state has an empty $\Psi$ and a single tree in $\Phi$ without any unchecked features. A parse fails when $\Psi$ is empty, $\Phi$ has multiple trees or unchecked feature bundles, and no operations are possible.

Tree nodes are like lexicon entries, maybe with some features checked. The form is $[\alpha\ \Gamma]$ if it is the node with word $\alpha$ closest to the root, or $\alpha$ if it is a lower node. A single node is also a tree.

At each step, the parser can perform one of three operations: MOVE a node on a tree to a higher position on the same tree, MERGE two adjacent trees, or shift a word from the input queue to the processing buffer in the form of a node with the corresponding features from the lexicon.

MERGE and MOVE are well known in the minimalist literature, but for parsing we present them a little differently. Their operation is illustrated in the subsequent examples. In this parser, MERGE finds the first[2] two compatible trees from the processing buffer with roots $\alpha$ and $\beta$, and replaces them with a tree with either $\alpha$ or $\beta$ as the root and the original trees as subtrees. MOVE finds a position $\alpha$ in a tree (including the possible future sister of the root node) that commands[3] a compatible subtree; the subtree is replaced by a trace at its original position and merged with the item at its new position. Adjunct movement and specifier movement are possible. Our examples only show adjunct movement. MOVE acts on the first tree for which this condition exists.

There are locality conditions for MERGE and MOVE. For MERGE, the trees must be adjacent in the processing buffer. For MOVE, the targeted tree positions must be the ones as close as possible to the root.

Compatibility is determined by feature checking. It relies on unification and unifiability tests. Sometimes checking succeeds without changing the checking status of its participants, because further checking may be required. These aspects of checking are also described in the examples.

At every step, MOVE is always considered before MERGE. This is to minimize the number of feature-compatible pairs of movement candidates within the trees in the processing buffer. If no movement is available in any tree, the parser looks for adjacent candidates to MERGE, starting from left to right. If neither MERGE nor MOVE is possible, a new word is shifted from the input queue.

### 3.3 Examples of Parsing

Here is the initial lexicon:

```
pater: unch(case:Nom, num:Sg, gnd:Masc)
filium: unch(case:Acc, num:Sg, gnd:Masc)
amat: {UNCH(case:Nom, num:Sg), UNCH(case:Acc)}
laetus: unch+(case:Nom, num:Sg, gnd:Masc)
laetum: unch+(case:Acc, num:Sg, gnd:Masc)
```

Inflection in Latin is often ambiguous. More entries for *laetum* would exist in a realistic lexicon. As disambiguation is not in the scope of this paper, we assume that the only entries in the lexicon are those useful for our examples.

[2] We scan from the left. Since the trees closer to the left end of $\Phi$ tend to have their features already checked, processing usually affects recently shifted items more.

[3] Node $\xi$ *commands* node $\zeta$ if $\xi$'s sister dominates $\zeta$.

**Example #1:** *pater laetus filium amat laetum*. This example illustrates the basic machinery of the parser, but it also demonstrates how the parser handles the discontinuous constituency of phrases, in this case noun phrases. *filium laetum* ("the happy son") is split across the verb. We begin with an empty processing buffer:

```
| pater laetus filium amat laetum
```

*pater* and *laetus* are shifted into the buffer one by one. They are "heard"; the lexicon is consulted and the relevant features are attached to them. (For the sake of brevity, we will combine multiple steps, particularly when a word is heard and some MERGE happens immediately as a result.)

```
[pater unch(case:Nom, num:Sg, gnd:Masc)]
[laetus unch+(case:Nom, num:Sg, gnd:Masc)]
| filium amat laetum
```

*laetus* adjoins to *pater*. There is no unifiability conflict between the features of both words. If a conflict had occurred, there would be no MERGE. The lack of conflict here indicates that the adjunction is valid. Thus, the feature on the adjective is marked as checked. Since adjunction is optional, and further adjunctions are theoretically possible, the feature on the noun is not checked yet.

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
      pater
      [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
| filium amat laetum
```

We shift *filium* into the buffer. *filium* cannot be absorbed by the *pater* tree. So we shift *amat*. Their nodes look as follows:

```
[filium unch(case:Acc, num:Sg, gnd:Masc)]
[amat {UNCH(case:Nom, num:Sg), UNCH(case:Acc)}]
```

Can anything be merged? Yes, *filium* checks a feature bundle on *amat* that was the one looking for another compatible bundle in order to project to the root of the new tree. When *filium*'s feature bundle checks with the corresponding bundle on *amat*, several things occur:

1. *amat*'s feature bundle's status changes from `UNCH` to `CH`, and *filium*'s bundle's status changes from `unch` to `ch`.
2. *amat* projects: a new tree is formed with *amat* and its features at the root. This is specified in the feature bundle: the capitalized form indicates that *amat* is looking for a constituent to fill one of its semantic roles.
3. *amat*'s feature bundle is unified with that of *filium* and replaced with the unification result; in other words, it acquires *filium*'s gender. *filium* does not gain any features, as its bundle is not replaced with the unification result. Only the projecting item *amat* is altered, as it now dominates a tree containing *filium* as a constituent. The non-projecting item becomes a subtree, inaccessible for MERGE at the root and thus not needing to reflect anything about *amat*.

Table 1 describes the interactions between feature bundle checking status types. In all four cases, only the item containing bundle 2 projects and forms the root of the new tree. A unifiability test occurs in each checking operation, but replacement with the unification result happens only in the feature checked on the projecting item (bundle 2) in the first case. No combinations other than these are valid for checking. The relation only allows us to check `unch` with `UNCH` feature bundles, since `UNCH` bundles indicate that their bearers project; there must be exactly one projecting object in any MERGE or MOVE. `unch+` check with `CH` feature bundles, because their `CH` (and feature-compatible) status will have resulted from a merge with an item that can accept the adjunct in question. `unch+` check with any compatible `unch` or `ch` feature bundles, as this indicates that the target of adjunction has been reached. We consider this analysis of checking relations to be exhaustive, but we save the rigorous elimination of other combinations for future work.

**Table 1.** Checking status interactions.

| **Bundle 1** | **Bundle 2** after checking: | **Bundle 1** | **Bundle 2** | **Replace bundle 2 with unif. result?** |
|---|---|---|---|---|
| `unch` | `UNCH` | `ch` | `CH` | Y |
| `unch+` | `CH` | `unch+` | `CH` | N |
| `unch+` | `unch` | `ch+` | `unch` | N |
| `unch+` | `ch` | `ch+` | `ch` | N |

Here is the result of the MERGE of *filium* and *amat* (to save space, we omit the *pater* tree):

```
([amat {UNCH(case:Nom, num:Sg),
              CH(case:Acc, num:Sg, gnd:Masc)}]
     [filium ch(case:Acc, num:Sg, gnd:Masc)]
     amat)
| laetum
```

MERGE occurs at the roots of trees. It treats each tree as an encapsulated object and does not *search* for places within trees to put objects. In more complicated sentences, searching would require more involved criteria to determine whether a particular attachment is valid. For the sake of minimality, we have developed a process that does not require such criteria, but only local interaction at a surface level. In doing so, we preserve our locality requirements.

The only attachments to trees that are valid are those that are advertised at the root. MOVE within trees takes care of remaining checkable features given criteria of minimality described above.

Now, *pater* is adjacent to *amat* and can thus MERGE with it, checking the appropriate bundle.

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
              CH(case:Acc, num:Sg, gnd:Masc)}]
      ([pater ch(case:Nom, num:Sg, gnd:Masc)]
            pater
            [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
      (amat
            [filium ch(case:Acc, num:Sg, gnd:Masc)]
            amat)  )
| laetum
```

In the processing buffer, no more features can be checked. The system needs to process *laetum*. It moves in and presents a problem:

```
[laetum unch+(case:Acc, num:Sg, gnd:Masc)] |
```

To what can *laetum* attach? We have defined the merge operation so that it cannot search inside a tree—it must operate on objects in the buffer. Fortunately, the rules we have defined allow an item with an adjunct feature bundle to be merged with another item with a `ch` projecting feature bundle if both bundles can be correctly unified. The adjunct feature remains `unch` until movement causes a non-projecting non-adjunct feature to be checked with it.

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
             CH(case:Acc, num:Sg, gnd:Masc)}]
      (amat
            ([pater ch(case:Nom, num:Sg, gnd:Masc)]
                  pater
                  [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
            (amat
                  [filium ch(case:Acc, num:Sg, gnd:Masc)]
                  amat)  )
      [laetum unch+(case:Acc, num:Sg, gnd:Masc)]) |
```

The rules for movement seek out the highest two positions on the tree that can be checked with one another. *filium* and *laetum* are precisely that. *filium* is copied, checked, and merged with *laetum*. For the sake of convention, we mark the original position of *filium* with a trace (<*filium*>).

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
              CH(case:Acc, num:Sg, gnd:Masc)}]
      (amat
            ([pater ch(case:Nom, num:Sg, gnd:Masc)]
                  pater
                  [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
            (amat
                  <filium>
                  amat)  )
      ([filium ch(case:Acc, num:Sg, gnd:Masc)]
            filium
            [laetum ch+(case:Acc, num:Sg, gnd:Masc)])  ) |
```

*filium* dominates because *laetum* is only an adjunct. All features are now checked, and the parse is complete.

**Example #2:** *pater laetus a filio laeto amatur*. This sentence is in the passive voice, included to demonstrate the need for feature paths. Passives in Latin are very similar to passives in English. *a filio laeto* is similar to an agent *by*-phrase. This requires new lexicon entries for all the words except for *pater* and *laetus*. The additional entries:

```
a: UNCH(case:Abl):right --> unch(by:0)
filio: unch(case:Abl, num:Sg, gnd:Masc)
laeto: unch+(case:Abl, num:Sg, gnd:Masc)
amatur: {UNCH(case:Nom, num:Sg, gnd:Masc), UNCH(by:0)}
```

The *by*-feature is similar to Niyogi's 2001 solution for *by*-phrases in English. 0 is there as a place-holder, since the *by*-feature does not have multiple values; it is either present or absent. We also use 0 to indicate that the feature *must* be present in order for the feature bundle to be unifiable with a corresponding `UNCH` feature bundle. *a* is a preposition in Latin with other uses (like *by* in English); making it necessarily attach to a verb as the deliverer of an agent requires a special feature.

The *by*-feature is the second element along a feature path. Before *a* can be merged with a verb, it must first be merged with a complement in the ablative case. This complement is directionally specified (to the right of *a*).

Let us begin:

```
| pater laetus a filio laeto amatur
```

*pater* and *laetus* are each shifted to the processing buffer. They adjoin:

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
      pater
      [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
| a filio laeto amatur
```

*a* and *filio* enter; *filio* is a noun in the ablative case and can be checked against *a*. (We will henceforth omit the *pater* tree until it becomes necessary.)

```
([a CH(case:Abl, num:Sg, gnd:Masc):right --> unch(by:0)]
      a
      [filio ch(case:Abl, num:Sg, gnd:Masc)])
| laeto amatur
```

*filio* checks the case feature on *a* immediately. There will be an adjective that needs to adjoin to *filio*, but it has not yet been heard; meanwhile, the system has a preposition and a noun immediately ready to work with. The mechanism of unification allows *a* to advertise the requirements of *filio* for future adjunction. There is no reason for the system to wait for an adjective, as it obviously cannot know about it until it has been heard. The noun does not need an adjective and only permits one if one is available.

As before, *laeto* is absorbed and attached to *a*.

```
([a CH(case:Abl, num:Sg, gnd:Masc):right --> unch(by:0)]
      (a
            a
            [filio ch(case:Abl, num:Sg, gnd:Masc)])
      [laeto unch+(case:Abl, num:Sg, gnd:Masc)])
| amatur
```

This adjunction occurs because unification and replacement have caused *a* to carry the advertisement for a `Sg`, `Masc` adjunct in its now-`CH` feature that previously only specified ablative case.

MOVE connects *filio* and *laeto*, leaving <*filio*>:

```
([a CH(case:Abl, num:Sg, gnd:Masc):right --> unch(by:0)]
      (a
            a
            <filio>)
      ([filio ch(case:Abl, gnd:Masc, num:Sg)]
            filio
            [laeto ch+(case:Abl, num:Sg, gnd:Masc)])  )
| amatur
```

The buffer now contains two trees (remember the *pater* tree). Neither of their roots have any features that can be checked against one another. *amatur* is heard:

```
[amatur {UNCH(case:Nom, num:Sg, gnd:Masc), UNCH(by:0)}] |
```

The case feature was checked on *a*, so the *by*-feature is available for checking with the verb. Recall that a MERGE in the processing buffer only occurs between adjacent elements. In the next step *amatur* can only merge with *a*.

```
([amatur {UNCH(case:Nom, num:Sg, gnd:Masc),
                CH(by:0, case:Abl, num:Sg, gnd:Masc)}]
      ([a CH(case:Abl, num:Sg, gnd:Masc):right --> ch(by:0)]
            (a
                  a
                  <filio>)
            ([filio ch(case:Abl, gnd:Masc, num:Sg)]
                  filio
                  [laeto ch+(case:Abl, num:Sg, gnd:Masc)])  )
      amatur) |
```

The *by*-feature on *amatur* has been unified with all the features on the feature path of *a*, required in case the order had been *pater laetus amatur a filio laeto*. In that situation, *a filio* would have been merged with *amatur* before *laeto* would be heard, since *laeto* is an adjunct. This mechanism ensures that permission for the attachment of an adjunct to *filio* is exposed at the root of the tree dominated by *amatur*. Here it is not an issue, but the operator covers this possibility.

Merging with *pater* (which we now reintroduce) is the next and final step:

```
([amatur {UNCH(case:Nom, num:Sg, gnd:Masc),
              CH(by:0, case:Abl, num:Sg, gnd:Masc)}]
      ([pater unch(case:Nom, num:Sg, gnd:Masc)]
            pater
            [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
      (amatur
            ([a CH(case:Abl, num:Sg, gnd:Masc):right --> ch(by:0)]
                  (a
                        a
                        <filio>)
                  ([filio ch(case:Abl, gnd:Masc, num:Sg)]
                        filio
                        [laeto ch+(case:Abl, num:Sg, gnd:Masc)])  )
            amatur)  ) |
```

## 4 Conclusions and Future Work

Through examples, we have presented an algorithm and formal framework for the parsing of free word order languages given certain limitations: highly constrained operations with strong locality conditions (MOVE and MERGE), no attempts at simulating nondeterminism (such as lookahead), and a limitation on the availability of the words in the sentence over time (the input queue simulates a listener processing a sentence as words arrive). Under these limitations,

we demonstrated the algorithm for a sentence with a discontinuous noun phrase and one in the passive voice.

The precedence of operations serves to connect shifted items semantically as soon as possible, though we do not fix the semantic representation; whenever MERGE and MOVE are performed, we assume that a more complete semantic representation of the sentence is achieved. We will seek compositional semantic formalisms that can handle the flexibility of our syntactic formalism.

The algorithm favours MOVE to ensure that all features in the current set of trees in the processing buffer are exhausted. This precludes multiple pairs of compatible subtrees, since that can only happen when MERGE joins a new item to a tree without exhausting the available movement candidates.

This has the effect of creating a "shortest move condition," which Stabler [2001] enforces by only allowing one instance of a pair of compatible features in a tree. Multiple compatible pairs can only arise when a movement frees a feature set along a feature path, and the feature bundles in the feature set are each compatible with different commanded subtrees. The highest commanded subtree moves first, since it is the first found in a search from the tree root. This exception is required by the mechanisms we use to handle free word order languages, which Stabler does not consider. We conjecture that this may only appear in practice if there are actual adjunct attachment ambiguities.

The locality of MERGE (adjacent items only) precludes search for every possible pair of movement candidates on the list. MERGE's precedence over shifting and locality together have the effect of forcing MERGE of the most recently shifted items first, since they bring in new unchecked features. (This fits the intuition that semantic connections are the easiest among the most recently heard items). Shifting last prevents occurrences of multiple compatible candidates for merging.

We will thoroughly demonstrate these claims in forthcoming work; here, we trust the reader's intuition that the definitions of these operations do prevent most ambiguous syntactic states, barring those caused by attachment or lexical ambiguities with semantic effect.

Feature paths are used to force checking certain feature bundles in certain orders, usually so that a word's complements can be found before it is absorbed into a tree. Feature sets allow the opposite; free word order languages let most features be checked in any order. Unification and unifiability tests provide sufficient flexibility for split phrases by allowing dominating nodes to advertise outstanding requirements of their descendants.

A less general version of this parsing algorithm, written in Prolog and designed specifically for these example sentences, showed that 90 of the 120 permutations of *pater laetus amat filium laetum* can be parsed. *filium laetus laetum amat pater* typifies the remaining 30, in that the adjective *laetum* cannot be absorbed by *amat*, since *amat* has not yet merged with *filium*; the 30 all have similar situations caused by the adjacency condition we impose on merging. These sentences are a subset of those exhibiting discontinuous constituency.

In a further experiment, we allowed implied subjects (for example, omitting *pater* is grammatical in Latin in example #1). This reduced the number of

unparsable sentences to 16. *pater* was still in all the sentences, but in the 14 that became parsable, *laetus* and *pater* were originally obstructed from merging. We lifted the obstruction by allowing *laetus* to merge with *amat* first. Without implied subjects, *laetus* acted as an obstacle in the same way as *laetum*; it ceased to be an obstacle after we introduced implied subjects.

Though we are able to parse most of the sentences in this constrained environment (and thus most examples of discontinuous constituency), we are working on determining how to minimally weaken the contraints or complexify the algorithm in order to handle the remaining 16 orders. But before deciding how to modify the system, we need to determine how many of these orders are actually valid in real Latin, and thus whether modifications to the system are really justified. We have embarked on a corpus study to determine whether these orders were actually plausible in classical Latin; a corpus study is necessitated by the lack of native speakers. We work with material generously provided by the Perseus Project (`http://www.perseus.tufts.edu/`).

This paper discusses the parser as an algorithm. In our pilot implementation, we simulated the checking relations between the Latin words that we used to experiment with the algorithm. We are now implementing the full parser in SWI Prolog using Michael Covington's GULP package to provide the unification logic. We will also seek a way to convert existing comprehensive lexica into a form usable by our parser, both for Latin and for other languages.

Work in minimalist generation and parsing has, thus far, mostly stayed within the limits of theoretical linguistics. A parser with the properties that we propose would help broaden the scope of the study of minimalist parsing to more realistic, complex linguistic phenomena. It could take this parsing philosophy toward practical applications. An example is speech analysis, where it would be advantageous to have a parsing algorithm that recognizes the need to make syntactic, and thus semantic, links as soon as a word enters the system.

## References


Michael B. Kashket. Parsing Warlpiri—a free word order challenge. In Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*. Kluwer, Dordrecht, 1991.

Sourabh Niyogi. A minimalist interpetation of verb subcategorization. International Workshop on Parsing Technologies (IWPT-2001), 2001.

Harm Pinkster. *Latin Syntax and Semantics*. Routeledge, New York, 1990. Translated by Hotze Mulder.

Edward P. Stabler. Derivational minimalism. In C. Rétoré, editor, *Logical Aspects of Computational Linguistics*, volume 1328 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1997.

Edward P. Stabler. Minimalist grammars and recognition. In Christian Rohrer, Antje Roßdeutscher, and Hans Kamp, editors, *Linguistic Form and its Computation*. CSLI Publications, Stanford, 2001.

Juan Uriagereka. *Rhyme and Reason: An Introduction to Minimalist Syntax*. MIT Press, Cambridge, Mass., 1998.