Greetings

Of course, let me first thank you for the invitation to Leipzig! Apart from the little contact we've had so far, this is the first time that our work gets public exposure in front of the research community, a fantastic chance for which we are very very grateful, as we think that the development of our system, while by no means finished, is reaching a point, where it could really contribute to the community. In the following hour we would like to introduce you to this evolving system, that could be of great help to researchers and students of the Latin language alike.

Our presentation will roughly cover the following points: Robert will fill you in about our own background, before we share with you our principles and design goals. What follows will be an overview of how our toolkit understands Latin, that is how it parses. This will include a closer look at data that was automatically produced when parsing the LDT Caesar text. We will go on with a short theoretical look into the future of the parsing sequence, before we come to the second big feature of our system, Creating Lating, where we will demonstrate some aspects of the easy-to-use interface the program provides, which enables us to create very interesting dynamic e-learning tools. in approximately an hour we'll reach a conclusion to this introduction, but first things first, I hand over the word to one of my team members and let Robert fill you in about our background.

Background

We usually don't like it when speakers talk so much about themselves before they start to get to the point, but in this case, we think an exception is imperative. We are no trained Computational Linguists - this fact dictates our process of work, as you will soon discover. Things are done in an unorthodox, different - and probably naive - manner, which might need some explanation. Filling you in about our own background will help you to understand some of the design choices we made.

The first idea to train a computer with Latin dates back to last year's September.

Robert and Gernot, both of us studied Latin Philology and Classical History in Graz - Robert is also studying to get a teacher's license for Latin and History, teach Latin in summer courses:

These are four-week long, totalling to 60 hours of contact between teachers and students, which attend these courses in order to receive their intermediate Latin certificate - the so called Latinum in Austria, I think it might be known under this name in Germany as well. The challenge as a teacher is that this is a highly compressed format, where students, which have never seen any Latin before, need to be able to translate for example Caesar after 4 weeks of training.

Due to the brevity of such courses we are constantly reviewing our own teaching process to find more efficient ways to present Latin grammar, that is: to find the bare minimum of grammatical knowledge that is needed for a human, to analyze and thus translate Latin texts. As we cannot hope that 4 weeks are enough to collect a good vocabulary, our focus in teaching has always been a correct and thorough morphological and syntactical analysis of texts, with a minimum of knowledge of word meaning. The course is usually split into two parts: The first half is teacher-centered and trains pure grammar, in the second half students try to use this knowledge to translate original texts with our guidance, but practically already on their own. We usually find that the step from schoolbooks to primary sources is a very difficult one, which causes many obstacles for students. What we wanted to have were exercises directly taken out from Latin authors, so that exposure to original texts starts from the very beginning of learning Latin.

Creating such exercises by hand appeared a bit tedious to us, we thought that this process could probably be automated by a system that can detect specific grammatical constructions, excerpt them to their relevant core - that means that you probably would want to leave out unneeded adverbials

when you just need to train AcI constructions - so the system should create exercises itself and should also be able to know the proper solution to those dynamically built training sets. This would not only enable us to create an almost infinite number of exercies, but would also drastically reduce the feedback loop for students:

They would not have to wait for a teacher's correction, as a computer system could immediately correct them or give hints to help the student find the solution on his own.

Of course, this is a task much easier said than done - we learned that soon after we started and it became quickly apparent that such a problem can only be solved when good parsing capabilities are provided, which opened also a new field for the project: creating treebanks for scientific research.

With such an idea in mind we contacted our third team member, Christof, who is an experienced and professional programmer and currently a student of Biomedical Engineering in Graz. We know each other for a very long time, he also introduced us to the world of Programming some time ago.

We didn't waste much time with thinking about the actual design and just started programming, starting as a private project.

The the whole system is built from scratch without any usage of existing libraries: We wanted to be indepedent from any third-party constructions and be able to realize our own design ideas - in hindsight this led to some problems, but came with the huge benefit of the possibility to find creative solutions for trivial and not-so-trivial problems without prejudiced opinions. Of course sometimes we were reinventing the wheel a bit here, but no harm done: we learned a lot in the process.

Timeline

The first code was created on the 8th of November. The first month was spent creating a morphology tool, after that we took a first - failing - attempt at syntactical analysis, which lasted roughly until January. That was also the time when we learned that the Perseus project had a treebank project already running, which was unknown to us until then. The code was completely reworked starting with late February/early March. Early May we contacted Gregory Crane with a very early and rough draft of some parsing results, who was so kind to forward our message to many people around the globe, which led to some great feedback for which we are very grateful. Since then development is still going strong, what you see now, will look different starting with tomorrow: everything is work in progress.

Principles and Requirements

Before we start to look at what the toolkit exactly has to offer, let us share some of our working principles and the requirements we impose on our systems. These are closely linked to our background and drive our development process.

Principles - our view of Latin

- Texts are written to be understood.
    - Generally authors will try to avoid ambiguities, as they wouldn't want to confuse their reading audience. Almost every sentence will have clear hints, that enable us to resolve ambigious meaning.
- A human is able to understand Latin of different genres and eras without *special training*
    - whether you read Cicero, Virgil or Thomas Aquinas, the same basic grammatical knowledge is applied.
- Semantics are not a deciding criterium

    - While semantics vary over the course of time, grammar is mostly stable. When we teach students Latin, we always tell them to try to solve a sentence without initially referring to the dictionaray - they rarely listen to us - the dictionary, which reveals word meaning, is most of the times not needed to analyze a text on a syntactical level. Morphological knowledge and syntactical grammar are the keys.
- Non-projectivity is nothing special
    - This is rewriting the past a bit - when we started we weren't even familiar with the term of projectivity, but the general concept was the same: non-projective edges are core to the Latin language - even more so to Greek as Francesco Mambrini and Marco Passarotti have shown in a recent paper about Non-projectivity in the Ancient Greek Dependency Treebank, but nonetheless - it's just a natural way of communicating in Latin should thus be enabled by default.

With such principles of Latin in mind, the following requirements were set-up that our system should be able to fulfill one day:

- The system should be two-way:
    - It should be able to **analyze**, as well as **construct** valid Latin - independent of author, genre or style.
    - Procedural rule appliance is thus ill-suited for the task, a context-free data-driven approach is as well.
    - It appeared to us that probably the easiest way for us to realize this, would be a more natural - a more "human" approach. When reading or writing Latin, especially if things are getting complex and the solution is not provided at first sight - humans use a top-down approach. Finding the predicate is usually the first step With a general knowledge of different "roles" in a sentence - be it subject, object or adverbial - a human scans the sentence looking at what's there and how he could fit the pieces together, so that a meaningful understanding can thus be achieved. This is the way we teach it, this is the way we know it best: and just like that, the system should be able to mimic our own thoughts.
- The program should be modular and highly susceptible to modifications and feature additions.
    - You should be able to use most of the system's tools independently, new knowledge should be able to be consumed by the program to enhance its qualities. For example a vast semantic layer - something the future might provide - should be easy to implement.
- Recalibrating the system should be easy
    - Most of the time we will know the source of a text. Even without a real scientific study a general knowledge about an authors style will be at hand. It's clear to everyone that Virgil uses non-projectivity a little bit different as Caesar for example. Different modes to help the program should be accessible to everyone with knowledge of Latin only, programming or computational linguistic skills should not be required.
- When parsing, no result is better than extremely malformed parsing trees
    - This might be controversial, but if our goal is creating data automatically, this data should generally be valid while avoiding false positives if possible, so that output can be statistically analyzed and visualized. Something like in this sentence should probably not happen, while this is quite ok.
- One language done well
    - While multi-language support (especially for Ancient Greek) would be welcome, one

language done well is the prime goal in the initial development cycle.

- The targeted computational-speed is "real-time"
    - Given a string of text, the system should be able to a solution without too much waiting time, using a regular consumer PC - that means that working with the system should be possible in a regular workflow.

It was apparent, that a higher-order programming language was the only possible choice: The domain - the Latin language itself - is complicated, precautions need to be taken to tackle code complexity that would sooner or later lead to a system, that is impossible to maintain. Because of its dynamism and object oriented approach, Ruby was chosen as a development platform. I know that this is basically just an implementation detail, that should play no role in this speech, but the decision to go object-oriented is a very important one, because it also led to grammatical implications: Every token will be an individual entity - an object, with distinctive behavior that determines the role it plays in a sentence.

These principles will reappear frequently in the following parts: Let's start the tour through the program, Understanding Latin - the parsing sequence

Understanding Latin

Tokenization & Morphology

The program starts with tokenization. The input string is splitted into tokens, as this example sentence from Caesar for example.

At this point the system already distincts between two kinds of behavior: word and punctuation – words in green, punctuation in red.

All words are analyzed, that means that the ending and all possible morphemes are separated from the word in order to get the supposed stem, which then is looked up in a database - our “stem-dictionary”.

In the case of our example sentence, the word “castra” will return four forms, that is four Word Objects, three of the noun castrum, one of the verb castrare. These two types differ in their behavior, while the verb castrare can take valencies, the noun forms can fill the role of a subject or an object. The word objects and their behavior are connected to the token “castra”.

Apart from morphological analysis and the appliance of behavior, phonological and metrical information can also be gathered at this stage. Every phoneme and every syllable gets evaluated. We’ll have a look on this feature later on.

I’d like to stress the fact that we really are concerned to collect as much information from our morphologizer-tool as possible, not just what the form is, but how it is constructed. This is important especially from a didactic point of view.

Syntactical Analysis

After tokenization and the morphological analysis,the interesting part - the syntactical analysis itself - starts. It can be split into 4 parts:

- Clause Finding
- Clause Analysis
- Clause Rating
- Clause Merging

What follows will be a rather general overview to get the concept across, which is basically quite

simple. Most complexity derives from some tricks to improve the computation speed, which shall not be discussed here.

As a first step the program tries to find the basic syntactic structure of a sentence. Just like a human would do, sentences are split and sorted , the main clause is detected, as well as subordinate clauses.

To achieve this, tokens - word as well as punctuation tokens - are wrapped, if possible, with a rough idea about the role they might play in their sentence. The word *Ubi* knows that it is a subjunction and can thus govern clauses. *Reduxit* and *faciunt* are easily distincted as well: both contain only one possible morphological form - and they both are verbs - that means they will have to play the role of something like a predicate in the sentence, we call them **safe verbs** at this stage. *castra* and *proelio* could be verbs as well, but are morphological ambiguous - they are **unsafe**.

Punctuation plays a role here. The program might even insert structuring punctuation, if it detects unmarked subordinate clauses.

The comma here is detected as closing a subordinate clause, in other cases a different role might be assigned. For example commas could serve in an enumerative role and would then be assigned coordinating behaviour.

The result of that process are clause objects which will serve as a container for the core piece of the parsing process.

Building Trees

Last week we learned that this process might remind some of the approach the Weighted Constraint Dependency Grammar is using - it shares some similarities, but is quite differently implemented. A so called TreeBuilder is laying out the field for the token objects, which drive the growing process of a tree - the leaves that are the result of this process should present an analyzed clause, where every token is used in a sensible way.

The TreeBuilder itself plays the role of something that could be described as gardener. He plants the tree's seeds and takes care of a well-arranged growing process.

The planted seeds are the Root nodes - there are several different: Roots for main clauses, subordinates, or elliptic clauses. These roots have behavior as well: they are destined for finding the first tokens, in most cases this comes down the predicate.

Thinking about the main clause in our sentence - proelio suos in castra reduxit the main clause root would find castra (the imperative), proelio (first person present) and reduxit as possible predicates - the leaves which choose castra and proelio as their predicate will soon be cut down, as there would be no possible use of reduxit left in such an event.

The leaf with reduxit as predicate will live on - and eventually come to a solution. These objects communicates with the pool of unused tokens and request forms that fit their needs. Each added form grows another leaf.

But let's take a step back. The root node requests a predicate. When reduxit is returned - the WordObject which was found by the Morphologizer - it is wrapped in another object. While the WordObject boasts its own general behavior, that will always apply, this new wrapper object is context-driven. The newly assigned behavior is bound to its surroundings - the other tokens present in the clause. The predicate itself is naturally very mighty, as at this point in time it's the only form used in the clause.

Reduxit requests its valencies and gets back word objects that will also be contextualized.

Tokens that get assigned at a later stage will be extended with a more constrained behavior: The idea is, that in every sentence every token has some kind relation to another - they affect each other and thus create different behavior. A quick example: *amicus militis* and *amicus miles Caesaris - The*

soldier's friend and *Caesar's friendly soldier*. Amicus derives from the same word in both instances. In *amicus militis* it's used as a substantive - substantives can take genetive attributes - and with such a behavior it will be extended with, if the form is called upon as substantive.

In *amicus miles Caesaris* amicus is an adjective attribute, congruent with miles. Adjective attributes cannot hold genetive attributes itself (these would belong to its parent, and that is exactly what happens here, Caesaris is an attribute to miles).

Because it cannot hold a genetive, it will not know of a genetive-taking behavior and will never ask for remaining genetives in the pool of unused tokens. Behavior is always dependent on its context, as information cascades trough the clause!

Every new token is itself able to grow new leafs, this process goes on until one - or in most cases several - complete leafs - using all tokens of a clause - are built, such a completed leaf can be seen on the slide.

One quick note: the process of finding new children is non-projective. We can calibrate the projectivity settings in close detail, allowing different projectivity modes for different word classes:

F.e. While we are very liberal with non-projective noun-to-adjective-attribute relations (which are cleary identified through congruence), we apply more strict rules for pure adverb-adverb relations. Different settings can be applied here and will play bigger role here when we open up the system for calibration through a web-interface, something I will mention in a couple of minutes again.

Back to our Leaf objects: When there are several possible solutions left, the "best" one needs to determined.

We use a weighting system in the form of ratings. These ratings try to be as general as possible, most of them are what we call "common-sense" rules and might not even be related to Latin as a specific language: They give no absolute picture about the clauses validity - validity is the concern of the tree-growth-process - ratings are a relative comparator between unique and valid syntactical solutions of a clause. An example rating: You see that Awards and Penalties are present. Ratings are relatively abstract and primarly concerned with determining grammatical features the sentence has.

Merging Clauses

We now have individual syntactically analyzed clauses - most of them with different possible solutions. Clauses are merged back to a whole sentence, subordinate clauses try to find hooks in superordinate clauses.

In the example sentence this is easily achieved, Ubi finds reduxit as hook and the subordinate clause will be used as adverbial. More complex situtations can occur of course. Merging is a another contextual layer to find the proper solution: Not the best leaf might be used, but the most appropriate. A relative clause with quod might have come to the conclusion that quod in the meaning of "because" would fit best, but looking at the big picture of a whole sentence, the usage as relative pronoun could be much better.

Evaluating Results

[this section refers to example sentences from the maltEval tool]

But just let's look at some results, a quick look at some real-world example will describe the system probably a bit better. There was a little challenge left before we could do that, as some of you might know, internally we use our own annotation system. This is a system that has been created in a generative process, just as the whole parser development, and follows a pragmatic style that suits our parsing style. From the get go it was planned though that output should be easy to transform, so that we can support different formats.

The XML conversion process is the first step in that regard, which was needed to compare our parser against the manually annotated LDT data. In most events this comes down to just switching labels, but there are some hard cases where additional logic is needed to get a valid XML result:

This is mainly a concern with prepositional objects. At the moment we annotate them just like that: as prepositional objects, but make no further distinction whether they serve an adverbial or object role. We hastily implemented a routine to create such labels, but it comes at no suprise, that this is by far the most frequent parsing error the evaluation - done through MaltEval as you can see - uncovers. Definitely an area where with a little more though we will do much better.

There are a couple of other little things that are mostly conversion errors: sentence adverbials in subordinate clauses like here, or the proper label for the participium coniunctum construction - where we do not modify the predicate. - you see another wrong prepositional object label here by the way.

Automatical parsing also uncovers how incredible tough it is to get a manual annotation right: This sentence should probably read like the automatic parsing „the army of the Roman people". I also like moleste as adverb better, but that might be debatable.

Another inconsistency where the automatic parser is much more determined concerns the proper head of subjects with coordinated predicates as can be seen in this example. I think it's a better solution to let both predicates govern the Haedui, which should be the agent for both verbs, if the Haedui led their troops into the territory of the Bellovaci and the Haedui begin to lay waste their fields.

Other cases are just subject to debate: "After he made a three days march" or "After he made a march for three days", attribute or adverbial accusative. The translation sounds better if you use the attribute, grammatically speaking you could probably make a case that this is just a regular adverbial accusative that measures time and modifies the predicate.

Another very good example is this sentence, much red here, but neither solution is really wrong. Robert will demonstrate that in our web-interface: The problem of the sentence basically comes down to a punctuation issue - how you would want to emphasize the predicates: When Caesar asked about their character and customs, he found out... or - it can be clearly seen if I insert more punctuation here: Caesar, when he asked, found out about their character and customs.

Of course, not everything looks as good as that, there are 4-5 trees which are badly malformed. It appears though, that most of them have something in common: they all contain numerals! That shows also one of the main caveats of our approach: If we don't teach our parser properly, he performs bad. The case of numerals can be explained: This test was the first time that our parser was confronted with Caesar - the main concept was mostly developed with Cicero.

Hw was chosen because of his defining character for Latin grammar, as well as being a highly-stylized orator, standing somewhere in the middle between easy prose (like Caesar) and tough poetry (like Virgil).

Now the problem is that Cicero doesn't really uses Numerals a whole lot, therefore we implemented them only very loosely at the moment and it cleary shows: much room for improvement here!

But things can get even worse: We mentioned earlier, that we do not build partial trees at the moment. When things go wrong no tree is built at all.

Let's look at this long sentence for a second. The subordinate clauses are parsed perfectly find, but the main clause terminates. We have only recently found the reason for this buggy behavior, the problem is, that we are too liberal: as you can see the main clause is an indirect speech with a long sequence of coordinated infinitives. We don't restrict such a dependency chain as you can see on the slide, where permittere is the head of consensisse, which is the head of coniurasse and so on.

If you factor in non-projectivty and all the coordination present, there is a nearly infinite number of

possibilities. The parser will stop because it runs out of memory. This is clearly a case of a missing constraint. Now that we have determined the cause, we might solve this problem soon. In the end, we had to take out three such sentences, because we get nothing back as a result. As was mentioned earlier, it's debatable to what degree that is a problem: At least we can clearly say that these sentences are wrong.

Quite often though things are looking like this: all green, everything right. 40% of all sentences are completely correct.

A preliminary overall score for the Caesar parsings looks like this:

84,7 for Labeld Attachment Score, 91 for unlabelEd attachment score (that is having the proper head), and 89,7 for Label Attachment (the correct relation label). This is already quite good, but chances are even better that these numbers will rise soon.

Improvements/Challenges

So far this is the state of the art regarding our parsing capabilities. We’re aware that there are still some challenges left, but chances for improvement as well. One of these challenges is Poetry

Poetry

Up to this moment we only parsed prose. But we’d like to prevent our system to be developed in only one direction, which may turn out as an obstacle to parse other genres as for example poetry. Thus we really need to start parsing poetic texts soon.

We have already seen that non-projectivity is no problem for our syntax analyzer, but to help the parser, we would like to take advantage of prosody.

Since we want to create a latin language toolkit, prosody should be a part of it anyway. We have already built two tools, which might help us here.

When you take a look at our screenshot here, you can see the the first of it, which evaluates the phoneme of a words, and the second one which evaluates the syllables.

Semantics

The implementation of semantics in our program is an outstanding goal, we’d like to achieve. We know that this is easier said than done. For the moment we use semantics only in a very marginal way. We have three semantic categories for nouns: animate, inanimate and abstract. The method to distinct these categories are very simple at the moment: At first, a distinction concerning the gender is made, every neutrum noun is inanimate. As a second step, we check for certain morphemes to categorize a noun. And last, there is a simple list of words, which contains animate nouns, which are not caught by the first two steps. For example the word homo is on that list. This is of course very, very elementary and not at all elaborated. But we really like to implement semantics, as I said, because in the end it may help the syntax analyzer to judge in certain cases, a quick example, the distinction between homographic forms of dative and ablative could get easier.

Database

Our system is highly dependent on a database, which contains, as mentioned before, the stem-dictionary. At the moment the analysis stops, when a word in the sentence is unknown to the stem-dictionary. This is a big problem, when parsing greater masses of text, because missing words interrupt our workflow: Either it leads to no result, or we have to handcheck if errors arose only because of an unknown word. For now our stem-dictionary is filled semi-automatically. The bigger part was filled with word lists from the net, which were formatted beforehand, a small part was typed in by hand. For the future missing word problems should be erased with a better

stem-dictionary.

Of course there are numerous challenges and possibilities to improve as our work is still in progress, but we'd like to leave it there for the moment.

Post-processing

This is only a theoretical idea at the moment, but will play a much bigger role in the future: Our parsing style has its benefits, but the data-driven approach has advantages of his own. Ideally such systems could be linked together, the one that is probably the easiest and will be tried in a first stage is using our system as postprocessor to data-driven parsings. The sequence could look like this: The data-driven parser creates many different parses, let's say they are xml, conll or whatever output format, our parser reads this output in and creates the Ruby objects it needs to work with the data. It could apply its own rules then - through constraints rule out absolutely impossible parsings (let's say a coordination binds a subject and an adverb), maybe even apply some ratings/weightings. The data would be reranked and a better overall parsing might be achieved. These thoughts shares the problem with the idea of general recalibration of our parsing tool. I cannot show you anything in that regard, but it is planned for the near future, that a user (us developers included) can change the parsers behavior: Apply different projectivity concepts or create a different weighting scheme. This could be done through a web-browser interface - I can probably talk a a bit more about those ideas at a later stage - outside of this presentation.

Let's get to the last stage of our presentation

Creating Latin

+++TERMINAL SESSION+++

The second part of our two-way-system is the construction of Latin words and syntax. Every functionality we apply to do parsing, can be inversed and be used to create Latin. We will show you this in a little terminal session, where you can also see the interface through which you can control the system. Let's start with the most basic token manipulation.

Christof creates new tokens out of a string of words and assigns them to a variable. Tokens can be created with the tokenize method, when we call it with the argument check: true we tell the system that it should look up the forms in the stem dictionary and do the morphological analysis.

What we get back is an array of tokens as you can see, we'll take pugnat at first, tokens.second will do the trick. Tokens contain word objects, in this case there is only one form of pugnat possible, we can select this form if we call the use method on a token, with the suggested number 1 we have the 3rd person present object selected.

Such word objects can change their form very easily, we can call

Back to the tokens the tokens array, let's select homo, as you can see, two word objects are present here, as homo could also be vocative apart from it's nominative form.

Select the homo form we can morph this one as well, or we could create congruences with other words: build congruences with method, takes a string or other word objects as argument, gives us back an array of congruence pairs

Select the first pair, this is completely morphable as well, for example let's raise the comparison level to superlativus. The forms change accordingly.

These simple changes build the base for more complex operations. Whole sentences can be also changed according to Latin rules,

Let's create a little sentence, "Uno tempore miles gloriosus ducem fortem necat." "At the same time

the glorious soldier killed the brave leader." Sophisticated message, sophisticated grammar. on this string we just call parse, what we'll get back is a Sentence object, fully parsed.

If we call the change method with an argument of how we want the sentence to be transformed, the system will use its Latin knowledge to do the operation: We could change the sentence to indirect speech f.e.

These are no simple string substitutions which would be to easy and would hinder us in using such an object further. As you can see, this is just a regular Sentence object with full functionality, where you could immediately create LDT styled XML for example

puts sentence.to_xml

These operations can be chained as well, if we invoke just another change call, because we want to change to sentence's voice, we pass the argument :passive and the transformation is executed, completely with building the proper passive agent of the former subject miles gloriosus.

Such an interface can now be used to dynamically create exercises.

This ability to create Latin can be used to create dynamic exercises.

[TERMINAL SESSION: CREATE DYNAMICAL EXERCISES]

The interesting point is not only that the exercise itself is created without much further ado, it's how the exercise behaves: wrong answers are parsed as well and the system tries to make sense about them - it tries to give feedback to the user, helping him solving the exercise. Tracking and parsing wrong answers opens up many new possibilities. Student could be "profiled" in the their learning process. The right exercise for the right student could be picked automatically, based on past performance of a student.

Such tools are in a very early stage, in fact, this is just a proof of concept and a look into the future. The point is, that with the system's interface such tools can be extremely easy built: The fill in the blank exercise, including the Hint Machine, was built on last Sunday, it took about 40 minutes to write the code. New exercise tools - more complex ones - can thus be implemented without too much hassle.

This almost ends our presentation, let us just add some concluding words.

Conclusion

To conclude this presentation, finally, let me make a personal statement. Last year’s work was enlightening and exciting and it presented a completely new view on Latin to us.

We think that these tools share a great deal of potential and we would like to see in a discussion how this could be exploited.

Thanks for your attention and patience.