Dissertação

Mestrado em Engenharia Informática - Computação Móvel

Framework to edit and use data from fieldwork in linguistic research

António Manuel Rodrigues Lopes

Leiria, *Setembro* de *2013*

Dissertação

Mestrado em Engenharia Informática - Computação Móvel

Framework to edit and use data from fieldwork in linguistic research

António Manuel Rodrigues Lopes

Dissertação de Mestrado realizada sob a orientação do Doutor Vítor Manuel Basto Fernandes, Professor da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria e co-orientação do Doutor Fernando Ramalho, Professor da Universidade de Vigo e membro do Centro Interdisciplinar de Documentação Linguística e Social.

Leiria, *Setembro* de *2013*

À Minha Família

Agradecimentos

Agradeço ao Doutor Vítor Manuel Basto Fernandes pela sua orientação, apoio e tempo dispendido durante o decorrer desta dissertação.

Agradeço ao CIDLeS – Centro Interdisciplinar de Documentação Linguística e Social, especialmente ao Mestre Peter Bouda pela sua orientação e apoio científico-tecnológico e ao Doutor Fernando Ramalho pela sua rápida disponibilidade, assim como aos restantes membros, pela oportunidade de investigação e todo o apoio dispensado durante o desenvolvimento deste trabalho.

Um especial agradecimento à minha mulher Joana Vieira por todo o tempo e ajuda na construção desta dissertação, assim como pelo imutável apoio também manifestado pela minha família à qual também agradeço.

Lince!

Resumo

O presente trabalho foi realizado no Centro Interdisciplinar de Documentação Linguística e Social, no âmbito do projeto *"Framework to edit and use data from fieldwork in linguistic research",* que é um dos sub-projetos pertencentes ao CLARIN (*Common Language Resource and Technology Infrastructure*) dentro do 7º Programa Quadro da UE (FP7). O objetivo do projeto consiste na construção de uma ponte entre os formatos de dados e ficheiros usados na documentação de línguas – especialmente *ELAN-Annotation-Format* – e os formatos de dados usados em corpus linguísticos e em Processamento de Linguagem Natural, NLP (*Natural Language Processing*), que hoje são armazenados em arquivos de línguas como o *The Language Archive* no Instituto Max Planck, situado em Nijmegen, Holanda. O projeto consiste no desenvolvimento de uma biblioteca que transforme os formatos de dados e ficheiros numa estrutura de dados unificados, seguindo uma implementação da ISO 24612 LAF (*Linguistic Annotation Framework*).

Palavras-chave: Linguistic Annotation Framework, Processamento de Linguagem, Corpus Linguístico

Abstract

The current work was carried out in *Centro Interdisciplinar de Documentação Linguística e Social*, within the project "Framework to edit and use data from fieldwork in linguistic research" which is one of the sub-projects within the CLARIN framework (Common Language Resource and Technology Infrastructure) within the 7th framework programme of the EU (FP7). The goal of the project is to build a bridge between data and file formats used in language documentation – specially ELAN-Annotation-Format – and the data formats that are used in linguistic corpus and Natural Language Processing (NLP) that are now stored in language archives such as The Language Archive at the Max Planck Institute in Nijmegen, the Netherlands. The project consists in a development of a library that transforms the data and file formats into a unified data structure according an implementation of the ISO 24612 LAF (Linguistic Annotation Framework).

Key-Words: Linguistic Annotation Framework, Language Processing, Linguistic Corpus

Índice de Figuras

Lista de Siglas

LAF	Linguistic Annotation Framework
GrAF	Graph Annotation Framework
NLP	Natural Language Processing
MASC	Manually Annotated Sub-Corpus
EAF	ELAN Annotation Format
TCF	Text-Corpus Format
HDR	Header File
TXT	Text File
TLA	The Language Archive
XML	eXtensible Markup Language

Índice

Introdução

A linguagem é um sistema de comunicação que desempenha um papel fundamental na metamorfose de uma sociedade e representa a raiz do pensamento e da expressão. Sendo uma ferramenta social, permite a interação de indivíduos dentro ou fora de um determinado grupo populacional, ajudando também, na compreensão da cultura e estilos de vida de outras comunidades. Para além disso, tem o poder de preservar o passado de uma cultura e transmiti-la sempre para as próximas gerações, através de registos escritos e falados.

"A diversidade linguística é essencial para a herança da humanidade. Toda e qualquer língua incorpora a sabedoria da unicidade cultural de um povo. A perda de qualquer língua é, portanto, uma perda para toda a humanidade." [1]

De acordo com os dados da *Ethnologue.com*[1] e do programa DoBeS[2] (*Dokumentation bedrohter Sprachen/Documentation of Endangered Languages*) iniciado pela Fundação V*olkswagen*[3] existem aproximadamente no mundo mais de 7000 línguas, das quais cerca de 2500 são línguas ameaçadas. Deste modo, conjetura-se que no final do século XXI só devam permanecer entre um terço a um décimo dessas línguas. Segundo o projeto da UNESCO[4] "Atlas of the World's Languages in Danger"[5], as principais áreas afetadas são a Índia (com 197 línguas ameaçadas), Estados Unidos da América (com 191), Brasil (com

[1] http://www.ethnologue.com/statistics, acedida 05/08/2013

[2] http://dobes.mpi.nl/dobesprogramme, acedida 05/08/2013

[3] http://www.volkswagenstiftung.de/en/foundation.html, acedida 05/08/2013

[4] http://www.unesco.org, acedida 05/08/2013

[5] http://www.unesco.org/culture/languages-atlas/index.php, acedido 05/08/2013

190), Indonésia (com 147), China (com 144), México (com 143) e a Federação da Rússia (com 131).

Durante os últimos 60 a 70 anos, mais de 200 línguas deixaram de existir. A extinção de línguas é um processo que está a aumentar drasticamente. Através do "Interactive Atlas of the World's Languages in Danger" da UNESCO é possível acompanhar esses números sendo as duas áreas mais afetadas, os Estados Unidos da América, com 54 línguas extinguidas, o Brasil e a Indonésia, com 12. Contudo, a sensibilização destes números, que têm vindo a diminuir à medida que o número de linguistas aumenta pelo mundo inteiro, e a revitalização realizada pelas comunidades de linguistas das consequências científicas e socioculturais das línguas ameaçadas conduziu à implementação e ao desenvolvimento da documentação da língua como sendo investigação autónoma dentro da linguística. Para além disso, os desenvolvimentos tecnológicos nos últimos anos contribuíram consideravelmente para a melhoria da documentação, a nível da recolha e armazenamento, e do processamento de dados sobre as línguas já armazenadas.

1.1 CIDLeS

A Documentação Linguística é composta por documentação, estudo e revitalização de línguas ameaçadas. Embora só tenha sido considerada e aceite como uma disciplina linguística por volta dos anos 90 do século XX, é uma área pela qual vários linguistas apresentam um certo interesse, especialmente aqueles que já contam com uma vasta experiência em trabalho de campo, como Bernard Comrie, Gipert Jost, Johannes Helmbrecht, Peter Austin, R.M.W. Dixon, William Croft, William Foley e Wolfgang Schulze.

O CIDLeS – Centro Interdisciplinar de Documentação Linguística e Social[6], centro onde esta dissertação foi realizada, foi fundado na freguesia de Minde em Portugal, tendo surgido com um objetivo focado essencialmente para a área da documentação linguística de línguas ameaçadas e minoritárias na Europa, associada a disciplinas como sociolinguística, etnolinguística, linguística antropológica, entre outras, bem como para a área da linguística tipológica e todos os seus aspetos. Para além do estudo, documentação e

[6] http://www.cidles.eu/, acedido 05/08/2013

disseminação de línguas ameaçadas e minoritárias Europeias, existe a parte devota ao desenvolvimento de tecnologias da linguagem para fins científicos e didáticos para as línguas ameaçadas e minoritárias. Com isto, espera-se também conseguir um novo ponto de vista sobre a investigação linguística praticada em Portugal, assim como, levar resultados portugueses para comunidades científicas internacionais.

1.2 Motivação

A sensibilização do número avassalador de línguas em vias de extinção levou à criação de grupos, centros e consórcios de investigação nas diversas áreas de estudo da linguística, principalmente na revitalização das línguas. Estes grupos têm essencialmente como objetivo o suporte financeiro para a documentação, disseminação e proteção de línguas ameaçadas e ou minoritárias. Dois exemplos são o ELF[7] (*Endangered Language Fund*) e o "Foundation for Endangered Languages"[8].

O projeto CLARIN[9] (*Common Language Resources and Technology Infrastructure*) é uma infraestrutura de investigação escolhido pela ESFRI[10] (*European Strategy Forum on Research Infrastructures*) *Roadmap*[11], que tem como propósito a coordenação de *networks* de instituições de investigação, universidades, bibliotecas e arquivos públicos, através de ferramentas e recursos tecnológicos que apoiem as ciências Humanas e Sociais.

Uma das infraestruturas pertencentes ao CLARIN é a parte alemã, CLARIN-D[12]. Esta contém vários grupos de trabalho, sendo um deles o "WG 3: Linguistic Fieldwork, Anthropology, Language Typology" [2], dedicado à linguística e à diversidade cultural do mundo, particularmente às línguas ameaçadas ou minoritárias. É neste sentido que o

[7] http://www.endangeredlanguagefund.org/, acedido 05/08/2013

[8] http://www.ogmios.org/, acedido 05/08/2013

[9] http://www.clarin.eu/node/3616, acedido 05/08/2013

[10] http://ec.europa.eu/research/infrastructures/index_en.cfm?pg=esfri, acedido 05/08/2013

[11] http://ec.europa.eu/research/infrastructures/index_en.cfm?pg=esfri-roadmap, acedido 05/08/2013

[12] http://de.clarin.eu/en/home-en.html, acedido 05/08/2013

CIDLeS, juntamente com o Departamento de Linguística e o Centro *eHumanities* da Universidade de Colónia, fazem parte do projeto com o título desta dissertação.

1.3 Objetivo da Dissertação

Esta dissertação tem como objetivo desenvolver e demonstrar a aplicabilidade da unificação das várias estruturas e formatos de dados, utilizados na documentação linguística e existentes no TLA[13] (*The Language Archive*) num formato único, LAF, através de uma biblioteca feita em linguagem de programação *Python*.

1.4 Organização da Dissertação

A presente dissertação está organizada em duas partes. Na primeira parte, constituída pelos capítulos 1 e 2, é feito o enquadramento teórico dos conceitos base para este estudo e na segunda parte, composta pelo capítulo 3, 4 e 5 é apresentado o desenvolvimento do projeto, os resultados da investigação empírica, assim como algumas sugestões.

No capítulo 1 é feita uma introdução à temática das línguas ameaçadas e minoritárias e ao centro de investigação (CIDLeS), indicando posteriormente a motivação, os objetivos do estudo para esta dissertação e apresentada por fim a organização da mesma.

No capítulo 2 é apresentado o estudo da arte em relação ao LAF, ao GrAF, às tecnologias da linguagem e seu desenvolvimento, bem como ao *The Language Archive*.

No capítulo 3 é apresentado o estudo, a arquitetura e metodologia do desenvolvimento da biblioteca Poio API.

No capítulo 4 são apresentados e analisados os resultados da investigação.

Por fim, no capítulo 5 é apresentada a conclusão do estudo e feitas sugestões para um trabalho futuro.

[13] http://www.mpi.nl/departments/other-research/research-projects/the-language-archive, acedido 08/08/2013

Processamento Linguístico

Na sequência da apresentação da motivação e objetivo do capítulo anterior, serão agora expostos alguns estudos sobre projetos e investigações científicas realizados, relativamente ao assunto das tecnologias da linguagem, da estruturação LAF (*Linguistic Annotation Framework*) e GrAF (*Graph Annotation Framework*) e do TLA (*The Language Archive*).

2.1 Tecnologias da Linguagem

As Tecnologias da Linguagem (TL) são cada vez mais uma parte integrante da interação computador-humano e, como tal, da vida quotidiana da maior parte das pessoas por todo o mundo. Deste modo, as TL encontram-se embutidas em quase todos os dispositivos eletrónicos, desde os dispositivos móveis de baixo custo, com a escrita automática T-9 (*Text on 9 keys*), até aos controlos de navegação por voz de carros de média e alta classe.

Desde a origem da era do computador que a linguística computacional e as áreas inerentes estiveram sempre entre os assuntos de maior importância, em parte devido às demandas das forças militares e policiais. Isto deveu-se essencialmente aos grandes investimentos em investigações científicas que foram feitos por entidades governamentais e privadas. No entanto, os resultados das pesquisas nem sempre cumpriram as metas esperadas pelos investidores e muitos ainda acreditavam que áreas como a tradução automática e o reconhecimento de voz ainda produziam erros controversos e nunca iriam ter qualquer usabilidade na vida real. É facto que algumas das expectativas criadas ainda não foram cumpridas e o mais provável é que permaneçam por cumprir durante mais algum tempo. Contudo, nos últimos anos ocorreu uma melhoria significativa na área do reconhecimento de voz, no qual, os sistemas modernos permitem atingir uma taxa de reconhecimento superior a 99% em tarefas de dicção.

Nos últimos anos as Tecnologias de Linguagem tiveram a oportunidade de usufruir de um vasto apoio financeiro, sendo a maioria das investigações focadas apenas nos sistemas que processam as línguas consideradas principais, tais como o Inglês, o Alemão, o Espanhol, entre outras, visto serem as mais faladas nos maiores centros económicos. No entanto,

estas línguas representam apenas uma pequena parte da diversidade global de línguas e são restritas a uma parte de uma subfamília de linguagem, nomeadamente, germânica, românica e ocasionalmente eslavo, quando pertencentes à família das línguas indo-europeu, o que leva a que a maioria dos sistemas estejam muito limitados no que diz respeito ao processamento de linguagem num sentido mais amplo. Isto tornou-se evidente com o processamento de "novas" grandes línguas, como o Árabe e o Chinês, que conduziram a grandes desenvolvimentos e a um progresso considerável neste campo de investigação, sendo ainda mais visível nas mais recentes abordagens da modulação computacional da unidade da língua na mente humana [3] [4] [5] [6].

Atualmente, cada vez mais se verifica que pelo mundo inteiro a tendência para a aprendizagem e uso de línguas, como o Inglês ou o Alemão, está inerente na sociedade, especialmente em meios de comunicação eletrónicos. A causa deste acontecimento deve-se fundamentalmente à falta e falha de *hardware* (como por exemplo o teclado) e de *software* (como por exemplo de transliteração e de complementação de texto) para as línguas ameaçadas e minoritárias. Isto acaba por restringir o uso natural da língua materna de cada indivíduo em diversas tarefas diárias, podendo mesmo levar à sua extinção.

Um dos grandes casos de sucesso dos dias de hoje e que tem contribuído, de certa forma, para a revitalização e divulgação de muitas "novas" línguas ameaçadas e minoritárias, é o projeto *The Endangered Languages Project*[14] da empresa *Google*, em que tem como objetivo disponibilizar o acesso *online* à documentação de línguas ameaçadas, criar espaços para os falantes das línguas e também partilhar conselhos e boas práticas entre aqueles que trabalham com as línguas ameaçadas [7].

Kevin Scannell, professor de Matemática e Ciência da Computação da Universidade de St. Louis, constitui também um grande exemplo do esforço que se tem vindo a fazer relativamente às Tecnologias da Linguagem para línguas ameaçadas e minoritárias, contando já com inúmeros projetos[15], dos quais são de salientar o "Accentuate.us"[16], que se

[14] http://www.endangeredlanguages.com/, acedido 03/02/2013

[15] http://borel.slu.edu/nlp.html, acedido 04/02/2013

[16] http://accentuate.us/, acedido 04/02/2013

trata de um teclado virtual que permite escrever com os caracteres especiais de mais de cem línguas, sem a necessidade da utilização de um teclado específico para cada uma delas, prevendo gradualmente as palavras com um processo similar à tecnologia T 9 [8] [9]; o "An Gramadóir"[17], que consiste num motor de verificação gramática em várias línguas, que pode ser associado a editores de texto como o *OpenOfficie*; e, por fim, o "IndigenousTweets.com"[18], em que o objetivo é conseguir construir comunidades *online* de línguas ameaçadas e minoritárias, fazendo uma pesquisa minuciosa por redes sociais como *Twitter* e *blogs* [10] [11].

As populações não devem apenas ser capazes de comunicar verbalmente ou por gestos na sua língua materna, devendo, por isso, aproveitar os grandes esforços que se têm feito para que as tecnologias estejam preparadas para as línguas minoritárias e ameaçadas, de forma a que as pessoas, através destas tecnologias, consigam fazer uma renovação da língua e da sua própria cultura, ou seja, fazer com que todos participem ativamente no ensino, na aprendizagem, no prorrogamento e na difusão da língua dentro das comunidades.

2.2 Objeto de Estudo – LAF e GrAF

Os corpora linguísticos são conjuntos eletrónicos de dados, usados essencialmente para o estudo e aplicações na área de Processamento de Linguagem Natural. Na maior parte das vezes, estes agrupamentos de dados estão anotados com informação linguística, como por exemplo, categorias morfossintáticas, estruturas sintáticas, entre outros, podendo ser usados para analisar correspondências entre eles, como por exemplo, em traduções paralelas.

O número de corpus linguísticos tem vindo a aumentar consideravelmente neste últimos 20-30 anos, o que conduziu à adoção de determinados princípios de representação como o uso de *stand-off annotation* [12] e XML, assim como ao desenvolvimento e tentativa de criação de mecanismos e formatos generalizados de anotação, tais como XCES[19] e *annotation graphs* [12] [13]. Embora já se tenha reconhecido que a comunalidade e a

[17] http://borel.slu.edu/gramadoir/index.html, acedido 04/02/2013

[18] http://indigenoustweets.com/, acedido 04/02/2013

[19] http://www.xces.org/, acedido 05/02/2013

interoperabilidade são cada vez mais inerentes para que se consiga fazer partilha, união e comparação de recursos linguísticos, o formato das anotações ainda é, na maioria das vezes, diferente de uma fonte para outra, parcialmente para se conseguir quebrar as barreiras impostas por determinados *softwares* de processamento linguístico. É neste sentido que surge o trabalho de investigação para a presente dissertação.

2.2.1 LAF

LAF - *Linguistic Annotation Framework* é uma ISO (*International Organization for Standardization*) com o código 24612:2012, em que específica "a representação de anotações linguísticas de dados da linguagem como corpora, linguagem gestual e vídeo" [14].

De acordo com Nancy Ide e Laurent Romary [15], foram identificados por especialistas os seguintes requisitos, durante o estudo do LAF [12]:

- Adequação Expressiva – Meios para representar todas as variedades de informação linguística, desde a mais generalizada até à mais detalhada;
- Independência Media – Manipulação e mecanismos de controlo para todos os potenciais tipos de media, como texto, áudio, imagem, vídeo, entre outros;
- Adequação Semântica – As estruturas de representação devem ter uma semântica formal, incluindo definições para operações lógicas;
- Incrementalidade – Suporte para vários estados para interpretação de valores de entrada e geração de valores de saída, bem como representações para resultados, quer parciais ou subespecificados, quer ambíguos, quer para a sua união e comparação;
- Uniformidade – As representações devem utilizar os mesmos blocos de códigos e métodos para que seja possível a sua combinação;
- Abertura – Não conter representações fixas, ou seja, dependentes de nenhuma teoria linguística;
- Extensibilidade – Possibilidade de declarar e fazer permuta de extensões, no sentido de centralizar o registo de dados da mesma categoria;
- Legibilidade humana – As representações devem ser facilmente legíveis, pelo menos na parte da criação e edição;

- Explicitação – A informação num esquema de anotações deve estar explícito;
- Consistência – Mecanismos diferentes não devem ser usados para indicar um mesmo tipo de informação.

Deste modo, considera-se que o modelo de dados LAF consiste numa estrutura [14]:

- Que descreve dados media, com o uso de âncoras que apontam para determinadas regiões nos dados primários;
- De gráfico, com nós, limites e ligações para regiões;
- De anotações para as representar e que pode conter uma estrutura de características.

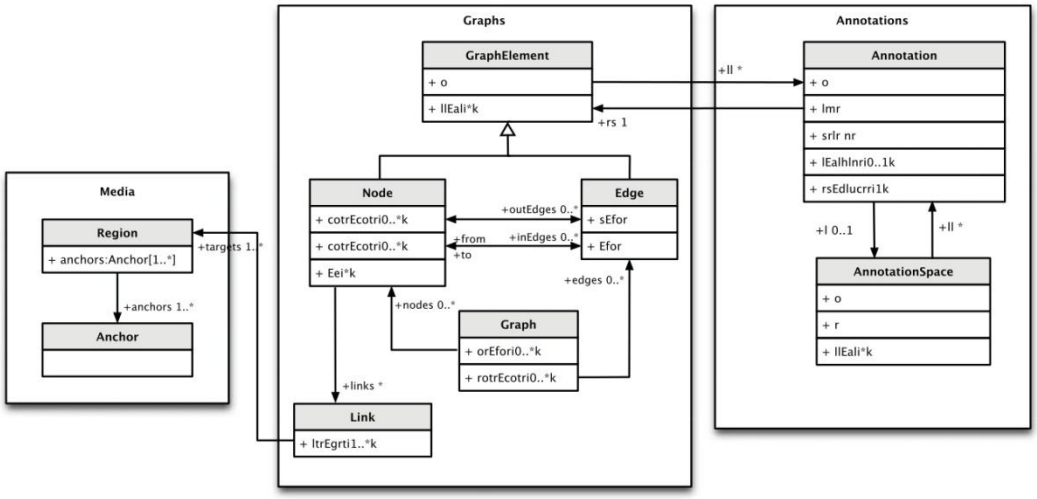

Figura 1 - Modelo de dados do LAF[20]

Por sua vez, a arquitetura do LAF consiste [14]:

- Num ou mais documentos de dados primários (*Primary Data*), que são compostos por dados eletrónicos em qualquer formato, tais como texto, áudio, vídeo, entre outros. De acordo com as especificações do LAF, são recursos que devem estar no modo de *read-only* (leitura), de forma a preservar a integridade das referências para as localizações dentro do(s) documento(s). Quaisquer correções e ou modificações nos dados primários serão tratadas como anotações e posteriormente armazenadas num documento de anotação à parte. Os documentos ou ficheiros que contenham dados primários em texto devem estar codificados em UTF-8 ou UTF-16. Na maior

[20] Figura proveniente do documento da ISO 24612:2012 [14]

parte dos casos, os dados primários não contêm qualquer tipo de *markup*, como por exemplo, *tags* XML. Caso existam, não deverá ser feita a distinção entre os *markup* e os restantes caracteres nos dados;

- Num número indiferente de documentos de anotação que contenham nós, limites e estruturas de características associadas a alguns ou a todos os nós e/ou limites de um grafo. Os documentos de anotação (*Annotation Documents*) contêm informação linguística relativa aos dados primários. As anotações são sempre associadas a um nó num gráfico, que está ligado às regiões definidas para determinados dados primários, podendo este estar ligado diretamente ou através de caminhos de nós alcançáveis. No âmbito do LAF, é recomendado que se faça a representação de cada camada linguística em documentos de anotação separados, para que seja possível fazer troca de informação. A granularidade de uma anotação, como por exemplo, a unidade da informação à qual a anotação se aplica, depende da aplicação de onde a informação é recolhida. Neste sentido, as anotações podem representar *tokens*, fonemas, palavras, frases, documentos ou simplesmente um corpus inteiro, mas também podem representar intervalos de tempo, incluindo unidades temporais como *timeslots e timestamps*;
- Num ou mais documentos a definir regiões que apontem para um documento de dados primários, que servirá como segmento base para as anotações. Estas referências diretas são feitas através da especificação de âncoras que ligam uma região. Na maior parte dos casos, estes nós ficam entre as unidades de base da representação dos dados primários. As regiões podem ser aplicadas em imagens de mapas, vídeos e áudio, sendo definidas com âncoras que especifiquem uma ou mais coordenadas, índice de *frames*, um ou mais pontos intermédios, entre outros aspetos;
- Num conjunto de cabeçalhos, incluindo um cabeçalho de recursos que descreve a coleção de documentos de dados primários e anotações, assim como cabeçalhos para cada documento de dados primários e anotações para as coleções. Mais detalhadamente, existem três tipos de cabeçalhos, ou seja, um cabeçalho de recursos (*Resource Header*), que descreve como um todo, incluindo a ele próprio, a estrutura e a codificação de um ficheiro e estabelece as definições que serão utilizadas no documento de dados primários, assim como nos cabeçalhos dos documentos de anotação; um cabeçalho do documento dos dados primários, que

fornece informação sobre a origem e os conteúdos dos dados primários, assim como especificar definições de categorias e tipos intermédios por referência a definições existentes no cabeçalho do recurso; por fim, um cabeçalho de documento de anotações, que inclui um subconjunto relevante de elementos do cabeçalho dos dados primários e que, juntos com outros elementos, fornecem ou apontam para informação relativa às categorias do conteúdo das anotações e das dependências entre documentos de anotações e outros documentos. Este cabeçalho não está num documento à parte mas sim no início de cada documento de anotações.

Como resultado final, a ISO 24612:2012 diz que o LAF consiste [14] [15] [16]:

- Num modelo de dados para anotações linguísticas e para os dados a que se aplicam;
- Numa arquitetura representativa dos dados da linguagem e das respetivas anotações;
- Numa serialização de XML para o modelo de dados, em que descreve uma estrutura de anotações de referência, associadas um conjunto de dados de linguagem, que se resume num gráfico ou gráficos diretos:
  - Os nós nos gráficos podem estar ligados às regiões dos dados primários;
  - Os nós e os limites podem estar associados a estruturas de recursos que descrevem propriedades sobre determinados dados primários.
- Sendo assim, o esquema de anotações deve ser isomorfo para com o modelo de dados do LAF.

2.2.2 GrAF

GrAF – *Graph Annotation Framework* é uma serialização em XML da estrutura genérica de grafos para anotações linguísticas, seguindo o modelo *Linguistic Annotation Framework* (LAF) [16].

Um documento GrAF/XML representa a estrutura de referência de uma anotação com dois elementos XML: <*node*> (nó) e <*edge*> (limite). Estes dois elementos podem conter uma etiqueta associada à informação de uma anotação. Tipicamente, as anotações que descrevem um determinado objeto são associadas a um elemento <*node*>. No entanto,

existe casos em que as anotações são classificadas como *edges* (limites), pelo que, quando isto acontece, o GrAF converte estes limites para nós para que se consiga fazer uma análise uniforme sobre os objetos anotados e as suas relações. A associação de anotações aos nós simplifica a relação existente entre uma anotação e vários objetos [16].

```
<edge from=”n1” to=”n2”>
<node id= “n3”>
```

As estruturas de características – <*fs*> (*feature structures*) – estão associadas a um determinado <*node*> e representam o conteúdo deste. Segundo as especificações da ISSO 24612:2012, estas estruturas devem conseguir definir herança, união e mecanismos de subsunção sobre as estruturas, permitindo representações de informação linguística de qualquer nível de complexidade [16].

```
<node id= “n3”>
   <fs>
     <f name=”annotation_value”>value</f>
   </fs>
</node>
```

Os elementos <*edge*> servem como ligações ou intervalos de regiões para outros nós, podendo não estar diretamente ligados aos dados primários. Para além disso, podem ser identificados com estruturas de características, mas a informação que consta nas referidas estruturas não representa uma anotação, como num nó, mas sim informação em relação ao significado ou papel da própria ligação [16].

Como base, o LAF utiliza a ideia dos Grafos de Anotação (*Annotation Graphs*) na representação dos dados linguísticos [13]. Os grafos geralmente podem ser vistos como um modelo de dados para as anotações linguísticas, dando uma visão geral e exemplos de como os vários formatos de dados provenientes de diferentes fontes, podem ser mapeados para a representação LAF através do GrAF, bem como a possibilidade destes grafos serem utilizados diretamente para tarefas de análise [16].

O GrAF já é o formato de publicação para *Manually Annotated Sub-Corpus* (MASC) da *Open American National Corpus* [17]. A *American National Corpus* (ANC)[21] fornece *plugins* para a *Unstructured Information Management Architecture* (UIMA)[22] – uma arquitetura e *framework* de *software* para criar, descobrir, compor e implementar um vasto leque de módulos com capacidades analíticas e interligá-los com tecnologias de pesquisa – e para *General Architecture for Text Engineering* (GATE)[23] – uma infraestrutura para o desenvolvimento e implementação de *software*, destinado ao processamento da língua humana. Deste modo, verifica-se a possibilidade dos dados e das anotações representadas pelo GrAF serem utilizados diretamente em sistemas de *workflow* científicos bem preparados [18].

A vantagem principal que se denota no uso do GrAF para os dados da documentação linguística é a sua abordagem de confronto radical, em que os dados e as anotações são completamente separados um do outro, podendo ser reunidos e melhorados de forma colaborativa num ambiente distribuído [19].

2.3 The Language Archive

O *The Language Archive* é uma unidade de trabalho dentro do Instituto Max Planck, localizado em *Nijmegen*, que foi desenvolvida desde os anos 90 com a finalidade de preservar dados linguísticos de todos os tipos, particularmente audiovisuais, que foram recolhidos durante trabalhos de campo nas comunidades de falantes de todo o mundo. Deste modo, o *The Language Archive* é composto por duas secções [20]:

- *Data Archive* (Arquivo de Dados), que é a parte responsável pelo armazenamento de material de línguas de todo o mundo, sendo gravada e analisada por investigadores das diferentes disciplinas da linguística. É possível ter-se acesso a este arquivo através de um *browser*, mas nem toda a informação é livre, por razões legais e éticas em relação à propriedade intelectual. De forma a manter o referido arquivo são cumpridos dois objetivos principais [21] [22]:

[21] http://www.americannationalcorpus.org/, acedido 10/08/2013

[22] http://uima.apache.org, acedido 10/08/2013

[23] http://gate.ac.uk/, acedido 10/08/2013

- “Manter o acesso a todos os recursos para a geração atual de investigadores, comunidades de línguas e público interessado” [22];
- “Preservar os valores importantes do património cultural para as atuais e futuras gerações” [22].

TLA Tools, que consiste na secção de sistemas da informação, que se encarrega por desenvolver e manter projetos de *software* que manipulam corpus linguísticos, desde a anotação ao *upload* de informação para o arquivo de dados [20].

Projeto de Software

Parte do objetivo desta dissertação foi o desenvolvimento de uma biblioteca que fizesse uma ponte entre os vários formatos de dados e ficheiros usados no processamento da linguagem. Este capítulo pretende mostrar as dificuldades e as soluções encontradas para que fosse possível a realização e conclusão do projeto, bem como a metodologia de desenvolvimento e as alterações na biblioteca *graf-python*, que é parte integrante do projeto.

3.1 Composição dos Ficheiros GrAF/XML

Existem duas formas de apresentar um GrAF, uma delas é através de um objeto orientado, composto por atributos e métodos, e a outra é através de ficheiros. No entanto, ambas partilham uma configuração similar na sua constituição.

Depois de ter sido explícita a importância dos *Document Headers* (cabeçalhos) e dos documentos – *Primary Data* (Dados Primários) e *Anntotation Document* (Documento de Anotações) – na utilização do GrAF, no capítulo "Processamento Linguístico", vai agora ser efetuada uma análise de como são compostos os ficheiros.

Segundo o LAF, é obrigatório que exista para cada documento de dados primários um cabeçalho em XML, que neste caso será o cabeçalho do documento dos dados primários (*Primary Data Document Header*) e deve ser criado como um ficheiro autónomo [14]. Este será o responsável por fornecer o *PID* ou a localização para um documento de dados primários e também toda a informação necessária para processar as anotações associadas a um determinado documento de dados primários. Deste modo, assume-se que este ficheiro seja lido em primeiro lugar quando um documento e as suas anotações estão para ser processados. Embora o cabeçalho de recursos (*Resource Header*) contenha toda informação relativa à estruturação e organização de todos os documentos presentes num modelo GrAF, o estudo sobre os cabeçalhos feito para esta dissertação incidiu apenas sobre o dos documentos dos dados primários uma vez que, segundo a ISO do LAF, este é suficiente e obrigatório para o processamento e transformação de dados.

Embora estes documentos tenham elementos e vários atributos, a especificação LAF diz que apenas alguns são obrigatórios e devem estar presentes sempre aquando da sua construção, por conseguinte, somente esses serão aqui expostos.

Começando pelos três cabeçalhos, a especificação LAF [14] defende que a estrutura XML de um *Primary Data Document Header* é a seguinte:

- *documentHeader* – nó principal (*root node*);
  - *fileDesc* – elemento responsável pela informação em relação aos dados primários, e contém os elementos XML, como o título e a origem do conteúdo dos dados;
  - *profileDesc* – elemento responsável pela informação sobre as características dos dados primários, como por exemplo, a língua usada, quem anotou o texto e, mais importante, a localização e nome dos documentos de anotações dos dados primários;
  - *revisionDesc* – elemento que vai conter informação relativa a alterações feitas no documento dos dados primários, assim como, nos seus documentos de anotação.

A figura 2 demonstra, de um ponto de vista generalizado, a estrutura do XML do *Primary Data Document Header*. Para além disso, apresenta um exemplo escrito.

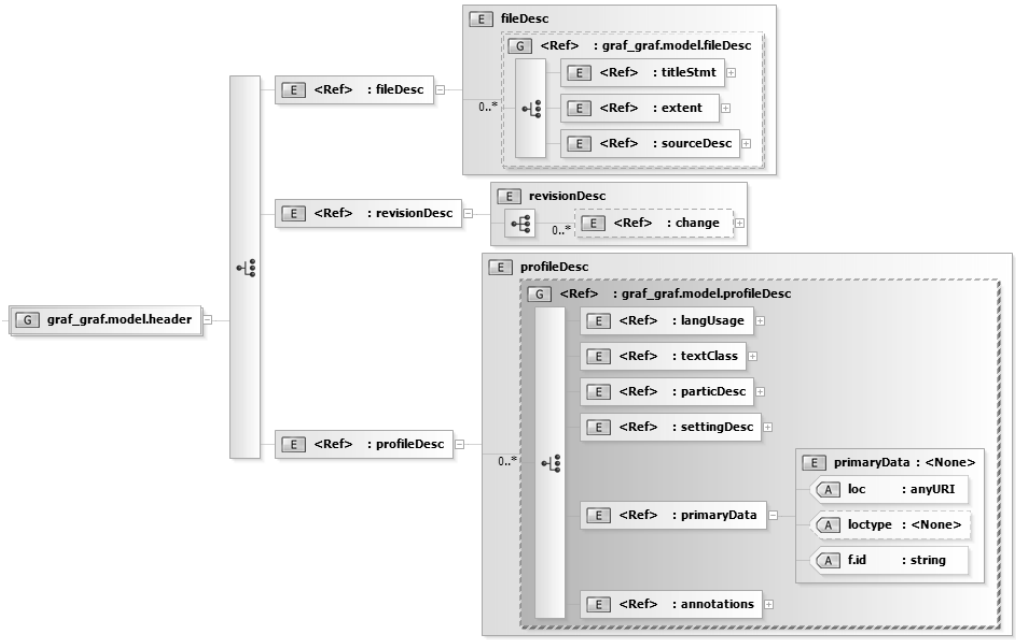

Figura 2 - Diagram da estrutura XML do Primary Data Document Header

```
<?xml version="1.0" encoding="utf-8"?>
<documentHeader creator="alopes" date.created="2013-07-17"
docId="PoioAPI-384232" version="1.0.0"
xmlns="http://www.xces.org/ns/GrAF/1.0/"
xmlns:xlink="http://www.w3.org/1999/xlink">
      <fileDesc>
                  <titleStmt>
                        <title>Text Example</title>
                  </titleStmt>
                  <sourceDesc>
                        <documentation>doc</documentation>
                  </sourceDesc>
      </fileDesc>
      <profileDesc>
                  <textClass catRef="EN"/>
                  <primaryData f.id="text" loc="text.txt"/>
                  <annotations>
                        <annotation f.id="words" loc="words.xml"/>
                        <annotation f.id="gestures"
loc="gestures.xml"/>
                  </annotations>
      </profileDesc>
</documentHeader>
```

O *Annotation Document Header* apresenta a seguinte estrutura XML [14]:

- *graphHeader* – nó principal (*root node*)
  - *labelsDecl* – elemento responsável por listar todas as *labels* (etiquetas) usadas nas anotações e por fazer contagens sobre a frequência de utilização;

- *dependencies* – elemento que contém os documentos que sejam necessários para a processamento das correntes anotações;
- annotationSpaces – elemento responsável por listar todos os grupos a que as anotações do documento pertencem.

Através da figura 3, é possível analisar, de um ponto de vista mais generalizado, a estrutura do XML do *Annotation Document Header*. É apresentada também uma demonstração.

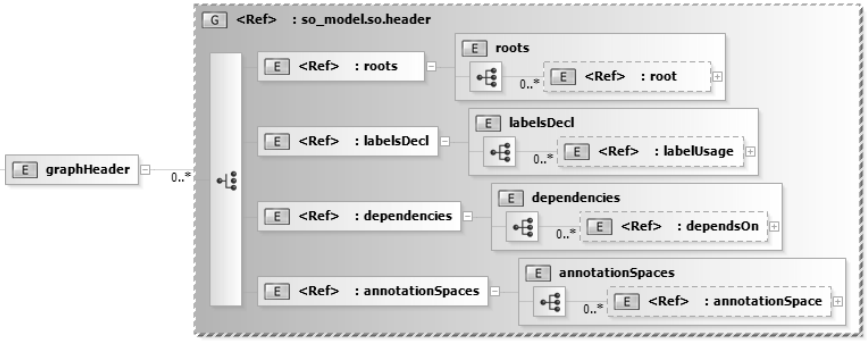

Figura 3 - Diagrama da estrutura XML do Annotation Document Header

```
<graphHeader>
            <labelsDecl>
                  <labelUsage label="words"
occurs="97"/>
            </labelsDecl>
            <dependencies>
                  <dependsOn f.id="utterance"/>
            </dependencies>
            <annotationSpaces>
                  <annotationSpace as.id="words"/>
            </annotationSpaces>
</graphHeader>
```

O LAF diz que os documentos de anotação (*Annotation Document*) devem utilizar a seguinte estrutura XML: um elemento <*graph*> que será o elemento principal (*root element*), constituído por um <*graphHeader*>; um <*node*> (nó); um <*a*> que representa uma anotação, uma <*fs*> (*feature strucutre*) estrutura de características, que será composta por <*f*> (*feature*) características, a qual pode apresentar várias estruturas de características (<*fs*>) em cadeia [14].

Analisando a figura 4, é possível verificar a estrutura do XML do *Annotation Document*, de um ponto de vista mais abrangente, bem como um exemplo escrito.

```
<?xml version="1.0" encoding="utf-8"?>
<graph xmlns="http://www.xces.org/ns/GrAF/1.0/">
      <graphHeader>
            <labelsDecl>
                  <labelUsage label="words" occurs="97"/>
            </labelsDecl>
            <dependencies>
                  <dependsOn f.id="utterance"/>
            </dependencies>
            <annotationSpaces>
                  <annotationSpace as.id="words"/>
            </annotationSpaces>
      </graphHeader>
      <node xml:id="words..W-Words..na100">
            <link targets="words..W-Words..ra100"/>
      </node>
      <region anchors="29370 29582"
xml:id="words..W-Words..ra100"/>
      <edge from="utterance..W-Spch..na19"
to="words..W-Words..na100" xml:id="ea100"/>
      <a as="words" label="words" ref="words..W-Words..na100"
xml:id="a100">
            <fs>
                  <f name="annotation_value">and</f>
            </fs>
      </a>
</graph>
```

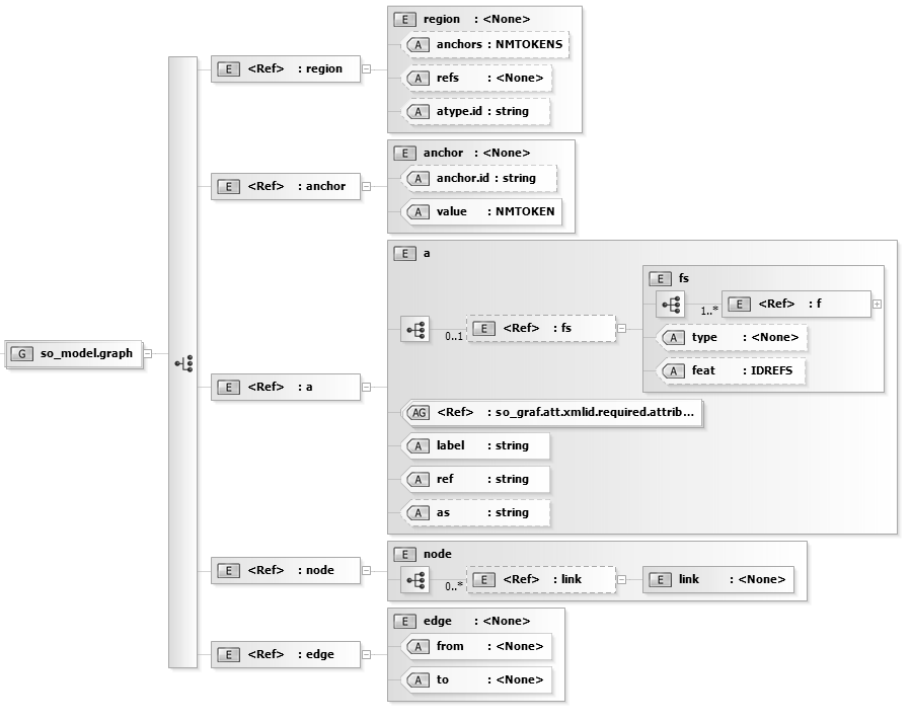

Figura 4 - Diagrama da estrutura XML do Annotation Document

3.2 graf-python

graf-python[24] é uma biblioteca livre e *open-source*, construída na linguagem de programação *Python*, no sentido de fazer o *parsing* (leitura) de ficheiros GrAF/XML, conforme a ISO 24612:2012 para grafos de anotação, assim como de transformar os grafos em ficheiros novamente. A biblioteca referida foi desenvolvida por Stephen Matysik, da *American National Corpus* (ANC)[25], e vai permitir representar a estrutura de grafos GrAF dentro da biblioteca Poio API.

Embora esta biblioteca não tenha feito diretamente parte do desenvolvimento da Poio API, serviu de objeto para o estudo efetuado nesta dissertação. Ao longo da evolução do projeto, o código existente foi sendo melhorado, tanto a nível de performance como de estruturação de código, conduzindo à implementação de novos métodos e classes. Inicialmente, foi feita uma otimização dos módulos, entre os quais se destacam o módulo *io*, responsável pelas classes e métodos de ações como *rendering* e *parsing* de ficheiros; o módulo *annotations*, que contém as classes relativas à construção de uma anotação; e, por fim, o módulo *graphs*, responsável pelas classes de representação e construção de um grafo ou objeto GrAF. Nas figuras 5, 6, e 7 é possível verificar os módulos respetivamente pela mesma ordem de ideias com mais detalhe através de diagramas de classes.

[24] https://github.com/cidles/graf-python, acedido 02/09/2013

[25] GrAF: The Graph Annotation Framework, http://www.americannationalcorpus.org/graf-wiki, versão na linguagem de programação Java, desenvolvido por Stephen Matysik

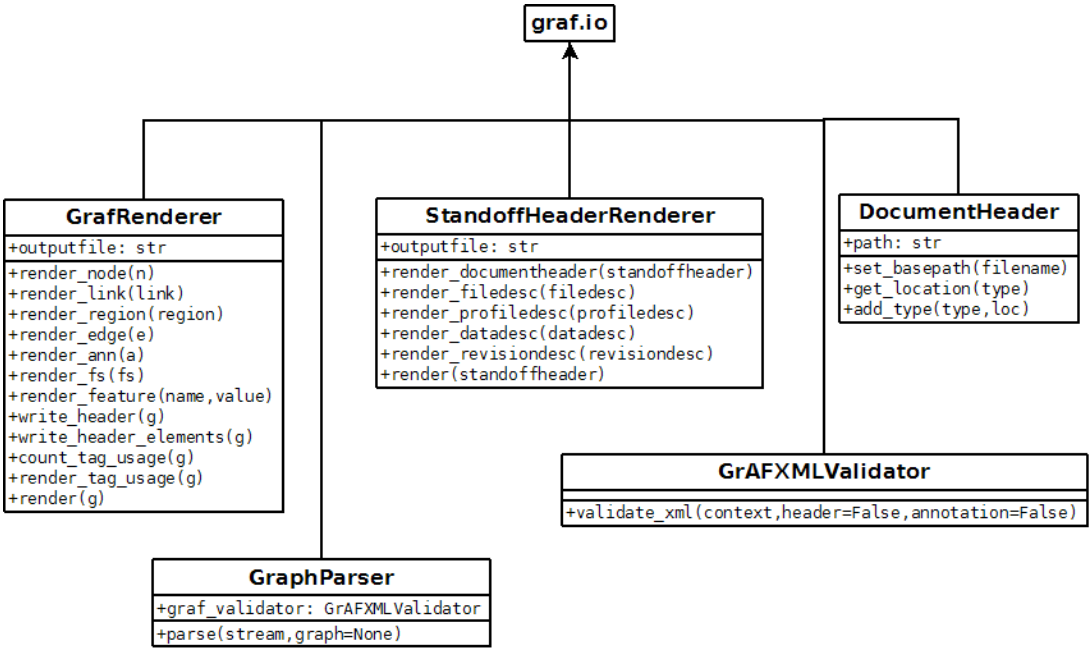

Figura 5 - Diagrama de classes do módulo io da biblioteca graf-python

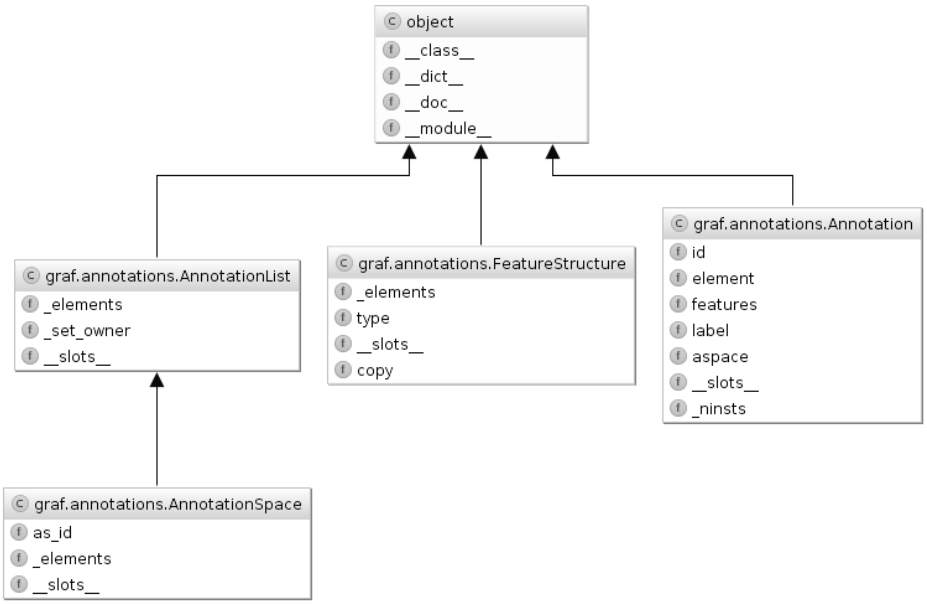

Figura 6 - Diagrama de classes do módulo annotations da biblioteca graf-python

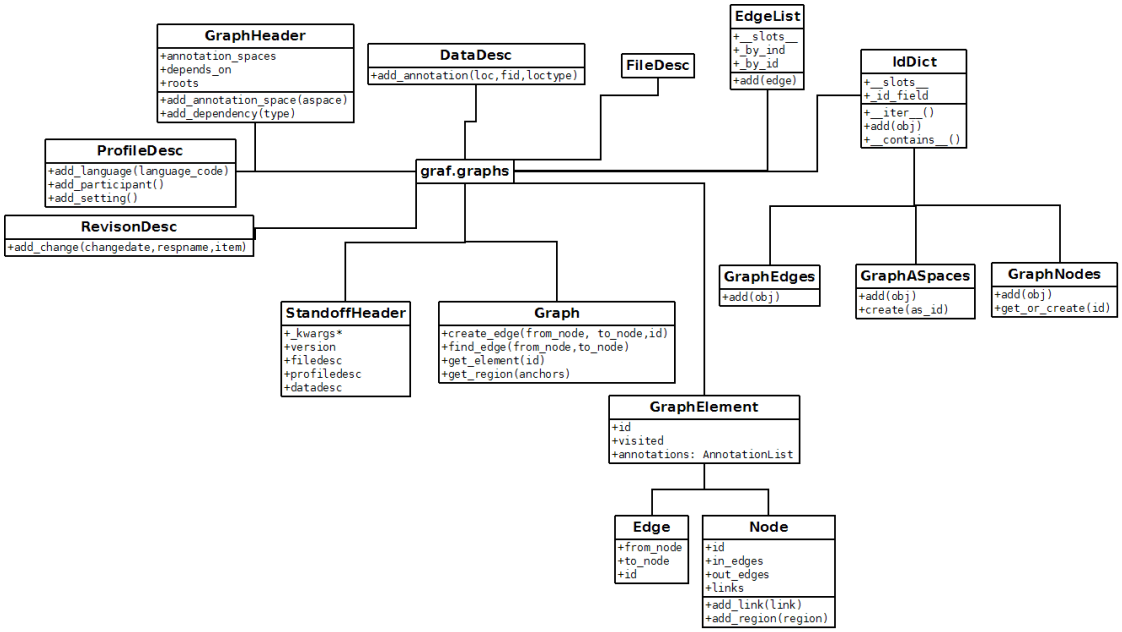

Figura 7 - Diagrama de classes do módulo graphs da biblioteca graf-python

Depois da reestruturação dos módulos foram realizadas alterações, mais especificamente no nome e organização dos elementos LAF para a versão mais atualizada, ponto que poderá ser analisado com mais detalhe na secção Composição dos Ficheiros/XML.

A fim de ser possível manter as mesmas especificações dos ficheiros GrAF/XML num objeto, foram criadas classes capazes de representar exatamente essas mesmas normas. No entanto, a representação do grafo – que inclui as anotações, nós, limites e regiões – já estava presente nesta biblioteca, pelo que apenas foram acrescentadas determinadas classes.

A classe *StandoffHeader* representa o objeto do cabeçalho dos documentos dos dados primários, composto por mais quatro classes: *FileDesc*, responsável por toda a informação relativa aos dados primários; *ProfileDesc*, responsável pela informação sobre o tipo de dados nos dados primários; *DataDesc*, responsável por armazenar a localização e tipo de dados primários, bem como o nome dos documentos de anotações a eles associados; *RevisonDesc*, responsável pelas datas sobre as alterações feitas nos dados primários e nos documentos de anotações associados. A figura 8 apresenta um diagrama de classes, com os detalhes das classes e com os atributos dos documentos XML que representam.

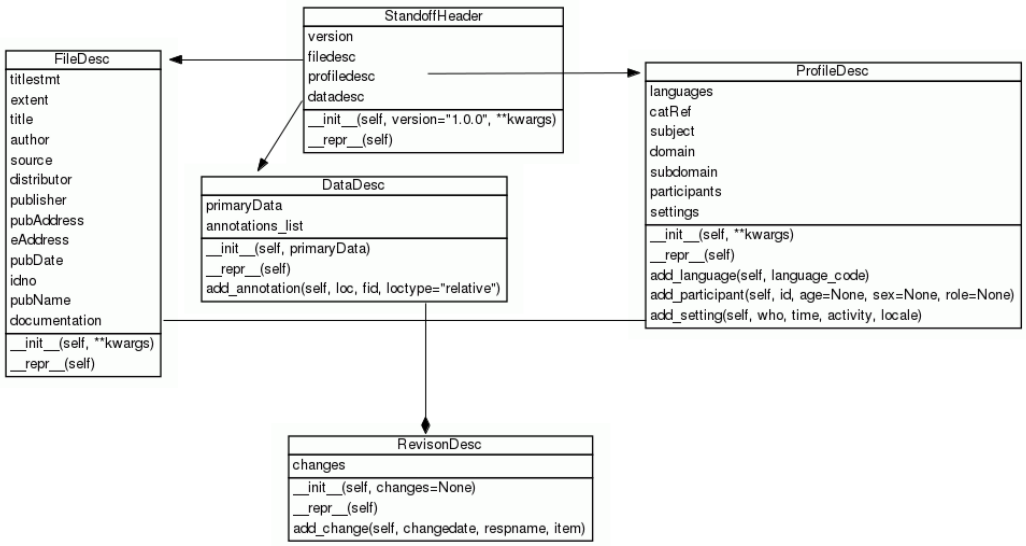

Figura 8 - Diagrama de classes StandoffHeader do módulo graphs da biblioteca graf-python

De acordo com o referido anteriormente, o módulo *io* tem a responsabilidade de tratar os valores de entrada e saída. Neste sentido, foram acrescentadas duas classes, por um lado, uma que ficasse responsável pela renderização dos objetos da classe *StandoffHeader*, com o nome *StandoffHeaderRenderer* e, por outro lado, outra classe com o nome *GrAFXMLValidator*, para validar os esquemas de XML para os documentos de anotações (*Annotation Document*) e o cabeçalho do documento dos dados primários (*Primary Data Document Header*). Esta classe tem como função garantir a coesão dos ficheiros renderizados a partir da biblioteca *graf-python*. A validação é feita segundo os esquemas XML (XSD) disponibilizados no *site* XCES para o GrAF[26].

Existem ainda duas classes dentro do módulo *io* que sofreram alterações, *GrafRender* e *GraphParser*. A classe *GrafRender*, encontra-se encarregue da renderização de um objeto GrAF para documentos de anotações, utilizava o processo normal de escrita de um ficheiro para transformar um objeto GrAF num ficheiro, o que levava a problemas de performance e condicionamento do utilizador à estrutura GrAF feita manualmente no código. Para isto, foi decidida a implementação da tecnologia *ElementTree*[27], que consiste numa estrutura em árvore de listas e atributos representados em dicionários, o que faz com que a utilização de

[26] Esquema XML do GrAF, XSD, http://www.xces.org/ns/GrAF/1.0/, acedido 12/08/2013

[27] http://docs.python.org/3.3/library/xml.etree.elementtree.html, acedido 12/08/2013

memória durante um *parsing* ou escrita seja muito menor. Deste modo, é possível a obtenção de uma melhor performance e usabilidade, visto que o objeto *ElementTree* fica responsável pela transformação do XML.

Para possibilitar o *parsing* de ficheiros GrAF/XML é necessário que os documentos de anotações de um determinado documento de dados primários estejam todos na mesma localização. O processo começa por fazer uma pesquisa em profundidade por dependências, ou seja, quando está a ser feita a leitura de um documento de anotação e este tiver dependências, será concretizada uma pesquisa dos documentos dessas dependências, sempre nesta ordem e como é exposto no fluxograma da figura 9. A legenda dos símbolos é apresentada no apêndice D.

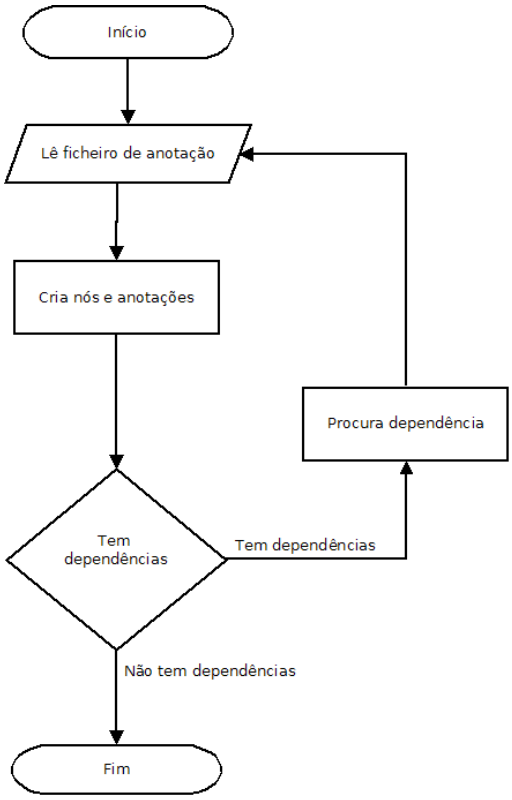

Figura 9 - Fluxograma sobre do funcionamento do antigo parser da biblioteca graf-python

As especificações do LAF defendem que para cada documento de dados primários deve sempre existir um cabeçalho e que este deve ser sempre o primeiro a ser lido durante o processamento de documentos de dados primários ou de anotações. Era esta falha que existia na classe *GraphParser*, em que o *parsing* dos ficheiros era feito apenas para um

documento de anotação, o que segundo as especificações não estava correto. Deste modo, o algoritmo de *parsing* foi alterado, no sentido de possibilitar a verificação do tipo de documento a ser analisado. Exemplificando, no caso de ser um cabeçalho – ficheiro com a extensão ".hdr" (*header*) –, o algoritmo procura pelos os documentos de anotação especificados no cabeçalho e procede à realização do *parsing*, tal como é demonstrado pelo seguinte fluxograma da figura 10. A legenda dos símbolos é apresentada no apêndice D.

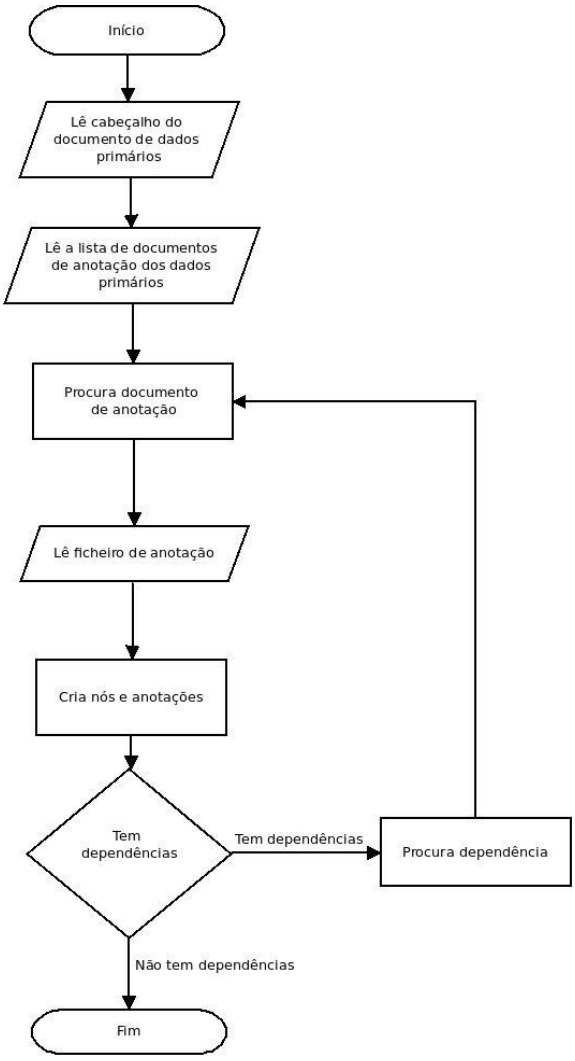

Figura 10 - Fluxograma sobre do funcionamento do parser atual da biblioteca graf-python

A figura 11 mostra a relação existente entre os cabeçalhos de um ficheiro GrAF/XML e o objeto GrAF, após a conclusão das modificações realizadas durante o desenvolvimento do projeto.

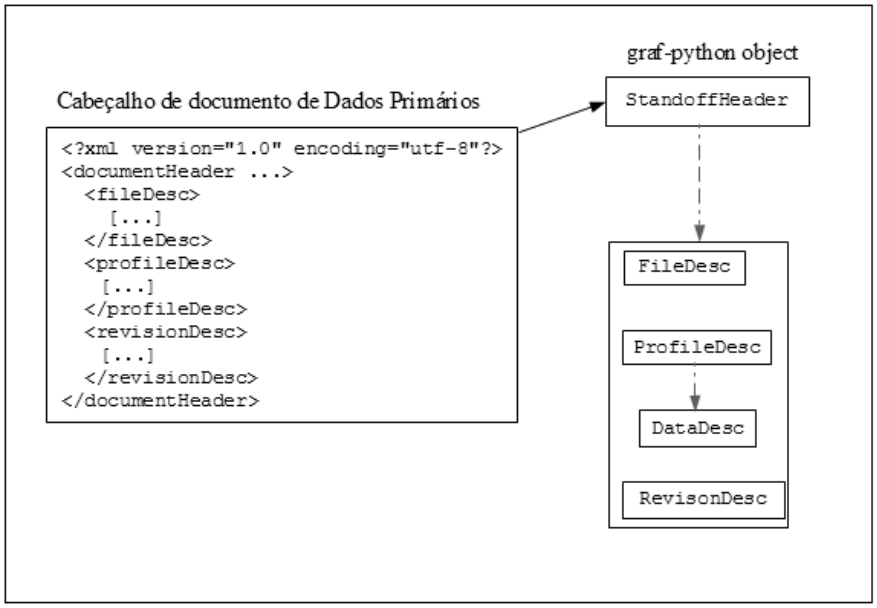

Figura 11 - Relação entre o objeto e ficheiros GrAF/XML

3.3 Poio API

Poio API[28] é uma biblioteca científica livre e *open-source*, desenvolvida na linguagem de programação *Python*, que permite aceder e analisar dados da documentação de línguas num *workflow* de análise linguística. Converte formatos de ficheiros, como por exemplo *ELAN-Annotation-Format* (EAF) [23], em gráficos de anotação como os especificados na ISO 24612:2012 do LAF. Estes gráficos usam a implementação GrAF que consente o acesso unificado de dados linguísticos de fontes distintas.

A Poio API é também a base para dois pacotes (*packages*) de *software*, Poio Editor[29] e Poio Analyzer[30], que são usados por investigadores em projetos de documentação linguística para a análise e edição de dados e anotações.

Esta biblioteca faz parte de um serviço *web* e de uma aplicação que permite aos investigadores acederem, pesquisarem e analisarem os dados guardados no arquivo de línguas – *The Language Archive* no Instituto Max Planck em *Nijmegen*, Holanda – assim

[28] https://github.com/cidles/poio-api, acedido 15/09/2013

[29] http://poio.readthedocs.org/en/latest/poiograid.html#introduction, acedido 07/11/2012

[30] http://media.cidles.eu/poio/poio-analyzer/, acedido 15/09/2013

como, dados locais ou dados provenientes de fontes que sejam desenvolvidas segundo a proposta *Weblicht* [24] do projeto CLARIN.

A implementação da Poio API é composta por três pontos essenciais:

- Representação interna LAF;
- Camada para um *Parser* e *Writer* para manipular os dados de diferentes tipos de formatos de ficheiro;
- Camada de *API* que forneça um acesso unificado para os dados da documentação língua através do uso de conceitos acessíveis para os investigadores.

A figura 12 mostra a arquitetura e como está inserida no projeto. O bloco maior, "*Library"* representa o Poio API. Este bloco contém no centro a representação genérica, LAF, que usa a implementação GrAF/XML, os "*Plugins"*, que são algumas das estruturas de anotações fixas já suportadas e, ainda, os mecanismos de "*Parser*/*Writer*" (leitura e escrita) e de pesquisa "*Search*". Os outros dois blocos, "*Web service*" e "*GUI - Desktop e Web*" (*Graphical User Interface*), são as camadas mais altas do projeto.

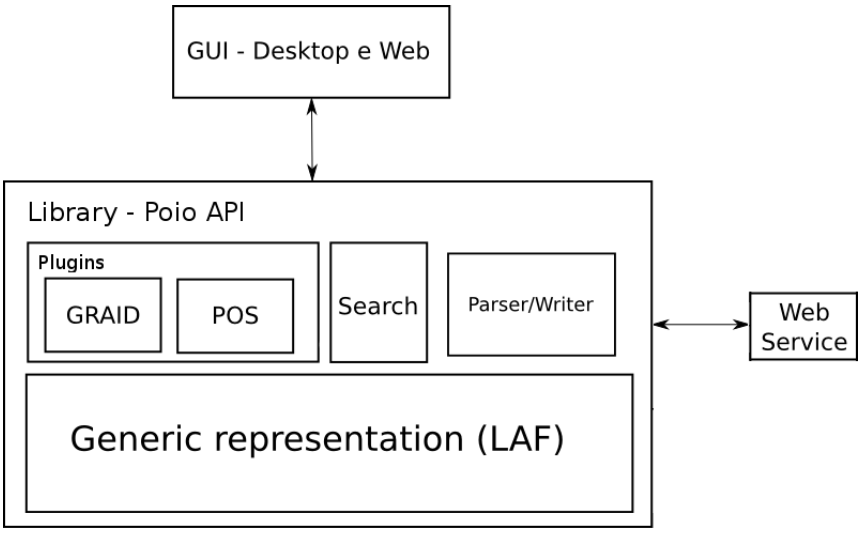

Figura 12 - Arquitetura da biblioteca Poio API

3.3.1 Data Structure Types

Na linguística geral e documentacional, os investigadores tendem apenas a criar esquemas de anotações baseados no que estudaram e normalmente estes são os que começam a usar nos seus projetos de investigação. Isto vê-se em concreto em *softwares* de anotação, como

o ELAN, em que se pode livremente criar camadas com um nome e ordem hierárquica especificadas.

Os esquemas de anotações podem ser muito versáteis, visto que cada linguista tem uma maneira específica de abordagem, pelo que seguem princípios muito diferentes uns dos outros aquando a anotação de um texto. Deste modo, surgem as *Data Strucutre Type*, que são tipos de dados que servem para fazer a representação de esquemas de anotações em árvore. Isto permite que um investigador consiga descrever de certa forma o mapeamento da estrutura das suas anotações perante um texto. Assim, consegue-se também obter esquemas hierárquicos e dependência entre os elementos que os constituem.

No exemplo abaixo é possível verificar uma simples *Data Structure Type* em que descreve que para cada *utterance* corresponde uma *translation*:

```
[ ' utterance ' , ' translation ' ]
```

Por sua vez, neste exemplo já se verifica que cada *utterance* pode conter uma ou mais *word* e corresponde uma *translation*:

```
[ ' utterance ' ,
     [ ' word ' ] ,
  ' translation ' ]
```

Um esquema um pouco mais complexo e que demonstra a potencialidade do uso destas estruturas, é o esquema de anotações GRAID (*Grammatical Relations and Animacy in Discoyrse*) [25], desenvolvido por Geoffrey Haig e Stefan Schnell, em que acrescenta o conceito *clause units* como uma camada intermediária entre as *utterance* e as *word* de cada uma delas. Com isto é possível ter noção dos possíveis e diferentes níveis entre os elementos:

```
[ '  utterance ' ,
     [ ' clause unit '  ,
          [ ' word ' , ' wfw ' , ' graid1 ' ] ,
```

```
      ' graid2 ' ] ,
' translation ', ' comment ']
```

Ainda existe o tipo de dados *AnnotationTree* que é o conjunto de dados das anotações, onde o conteúdo é armazenado numa estrutura em árvore que espelha a hierarquia de uma *Data Structure Type*. A figura 13 mostra a relação entre a *AnnotatioTree* e a *Data Structure Type* e também se verifica que o parênteses reto "[" significa que existe um ou mais valores para aquele elemento.

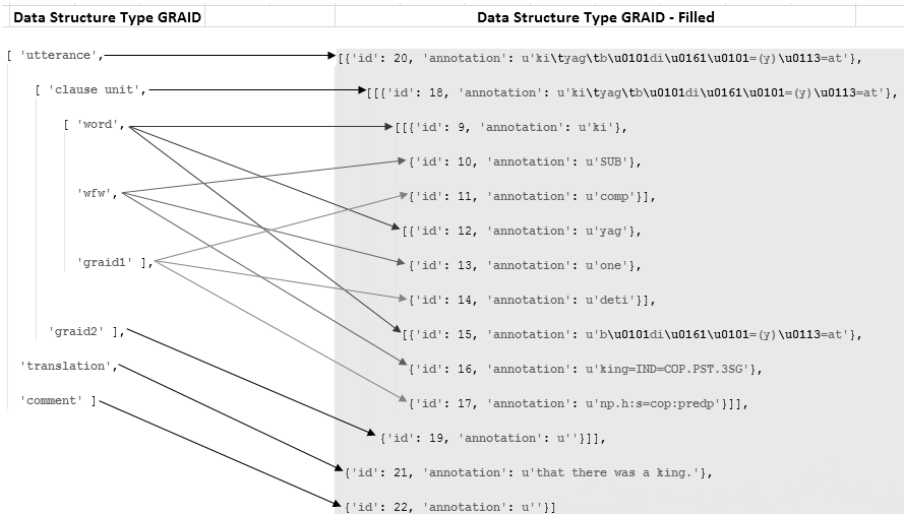

Figura 13 - Representação de uma Annotation Tree

Como foi possível ver nos exemplos anteriores a hierarquia e a dependência de elementos é atribuída com o uso dos parênteses retos "[ ]". O nivelamento da hierarquia é feito com base na posição do elemento sempre que se inicia uma nova lista, isto quer dizer que o elemento primordial de cada nível é sempre o primeiro elemento de cada lista, mesmo quando um dos elementos é uma lista, como é demonstrado através da figura 14:

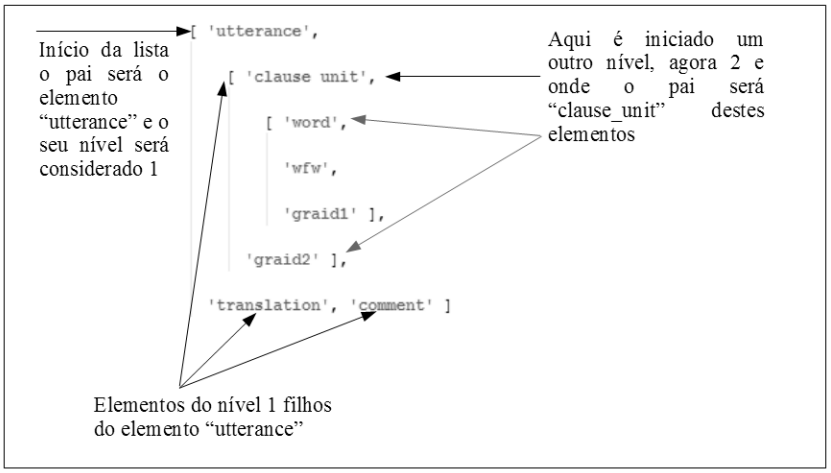

Figura 14 - Esquema hierárquico de uma Data Structure Type

As duas vantagens encontradas na utilização destas árvores para a representação de esquemas de anotações são, primeiramente devida ao facto dos linguistas conseguirem facilmente entender o seu funcionamento e criar os seus próprios esquemas de anotações; e, por outro lado, a possibilidade de transformar diretamente estas estruturas para os *softwares* de edição e análise de anotações, Poio Editor e Poio Analizer [19].

Poio API utiliza apenas um subconjunto das potencialidades dos grafos GrAF para a representação das anotações baseadas em camadas. Neste sentido, significa que apenas vai existir a preocupação em criar certos e determinados limites entre os nós e as suas anotações, de modo a representar a relação pai-filho entre as anotações que estão em camadas diferentes no ficheiro de anotações original, ou seja, não serão criados limites entre anotações da mesma camada nem entre anotações sem relação pai-filho. Além disso, existe um elemento fixo para todas as estruturas de caraterísticas denominada de *annotation_value*, que corresponde na maior parte dos casos ao valor dos dados primários referentes à anotação.

Para representar os dados das anotações baseadas em camadas, a biblioteca Poio API utiliza internamente a biblioteca *graf-python* para armazenar os dados e as anotações. Esta estrutura padrão de dados GrAF consiste num conjunto de nós (*nodes*) e limites (*edges*) enriquecidos por estruturas de características (*features structures*) que contêm as anotações linguísticas. Os limites (*edges*) são o elo de ligação entre nós individuais que contêm uma relação comum para com uma anotação, geralmente numa hierarquia superior, e

normalmente com estruturas de características semelhantes. Os nós estão geralmente associados aos dados primários, como por exemplo, texto ou áudio, através de regiões (*regions*). Contudo, podem existir nós que não estejam diretamente ligados aos dados primários e apenas representem anotações ou sub-anotações de um outro nó. A figura 15 demonstra como um nó é representado:

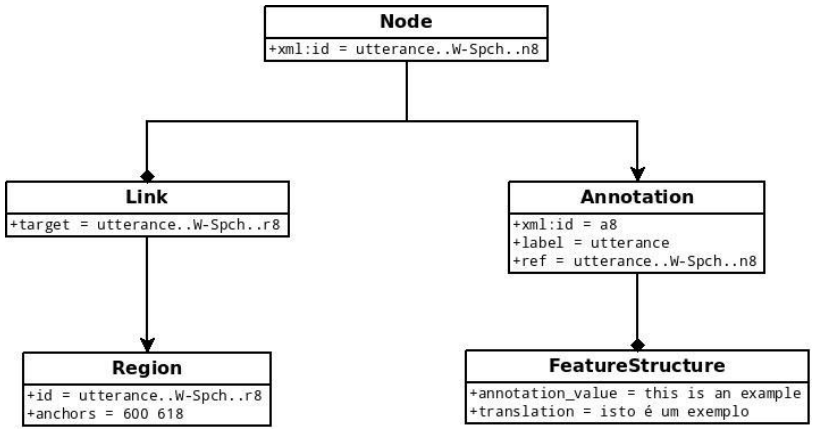

Figura 15 - Representação interna de um nó na biblioteca Poio API

3.3.2 Implementação Técnica da biblioteca Poio API

Nesta secção serão expostos com maior detalhe a organização da biblioteca, assim como o estudo e a sua composição e ainda os obstáculos enfrentados durante a sua conceção.

3.3.2.1 Organização dos Módulos Poio API

A biblioteca Poio API é composta por 3 pacotes (*packages*) de módulos, onde o principal, designado "poioapi", é constituído por:

- *annotationgraph* – módulo que permite a manipulação dos dados de anotação de um GrAF através de duas classes essenciais:
  - *AnnotationGraph(data_structure_type)* – classe responsável pelo armazenamento dos dados de anotação em grafos de anotação, de alguns formatos de dados, tornando-os acessíveis através de estruturas hierárquicas de camadas;
  - *AnnotationGraphFilter(annotation_graph)* – construtor de uma estrutura em árvore, que possibilita a pesquisa de informação num objeto Graf.
- *annotationtree* – contém as classes para aceder aos dados anotados no formato de

annotation trees;

- *data* – construtor de objetos do tipo *Data Structure Type*, a partir dos vários formatos de dados existentes, que contém as classes e métodos para acederem e manipularem os dados e estruturas extraídas.

O segundo *package*, "io", é responsável pelo tratamento dos dados de entrada e saída, com os módulos:

- *brat* – contém as classes para transformar objetos GrAF no formato *brat stanf-off annotation* [26];
- *elan* – contém as classes para aceder aos dados do tipo de ficheiro usados pelo *software* ELAN[31] e também para escrever objetos GrAF e *Elan Annotation Format* (EAF);
- *graf* – contém os métodos responsáveis pela leitura e escrita de ficheiros ou objetos GrAF;
- *pickle* – contém os métodos responsáveis por ler e escrever ficheiros GrAF/XML de ficheiros *pickle*[32] que usam dados *Annotation Tree*;
- *tcf* – contém as classes para aceder aos dados do tipo de ficheiro *Text Corpus Format* (TCF) [27] [28], usados no projeto *Weblicht* e escrevê-los em objetos GrAF;
- *toolbox* – contém as classes para aceder aos dados do tipo de ficheiro usados pelo *software Toolbox*[33] e escrevê-los em objetos GrAF;
- *typecraft* - contém as classes para aceder aos dados do tipo de ficheiro usados pela ferramenta do projeto *TypeCraft* [29] e escrevê-los em objetos GrAF.

Para além dos expostos, existe ainda o *package* "tests", que é o responsável pelos testes de desenvolvimento aplicados aos módulos anteriores.

3.3.2.2 Visão Geral de Desenvolvimento da Representação GrAF

Depois de algum estudo e seguindo as normas do LAF concluiu-se que seria suficiente para a construção de um objeto GrAF, através do *parsing* (leitura) dos distintos e vários

[31] http://tla.mpi.nl/tools/tla-tools/elan/, acedido 08/08/2013

[32] http://docs.python.org/3.3/library/pickle.html#data-stream-format, acedido 08/08/2013

[33] http://www-01.sil.org/computing/toolbox/information.htm, acedido 08/08/2013

formatos existentes, a utilização de um algoritmo que aquando de lhe ser enviado um objeto *parser*, conseguisse ter acesso à informação relativa às camadas principais. De seguida, recorre-se a um ciclo, no sentido de percorrer todas as camadas para encontrar os seus filhos e as suas anotações. Dentro do ciclo, para cada anotação com uma função recursiva, vão encontrar-se as anotações para os seus filhos e assim sucessivamente. À medida que se vai encontrando uma anotação nova é-se atribuída ao grafo. Segue-se o pseudocódigo para análise e a figura 16, na qual é apresentado o fluxograma do algoritmo de conversão. A legenda dos símbolos é apresentada no apêndice D.

```
Função converter_camada(camada)
    início
        filhos_camada = devolve_filhos_para_camada(camada)
        anotações = devolve_anotações_para_camada(camada)
        para anotação em anotações fazer
            início
                adicionar_no_ao_grafo(anotação)
                Se filhos_camada > 0 então
                    início
                        para filho em filhos_camada fazer
                            início
                                converter_camada(filho)
                            fim
                    fim
            fim
    fim

Início
    para camada até todas_camadas_principais fazer
        início
            converter_camada(camada)
        fim
Fim
```

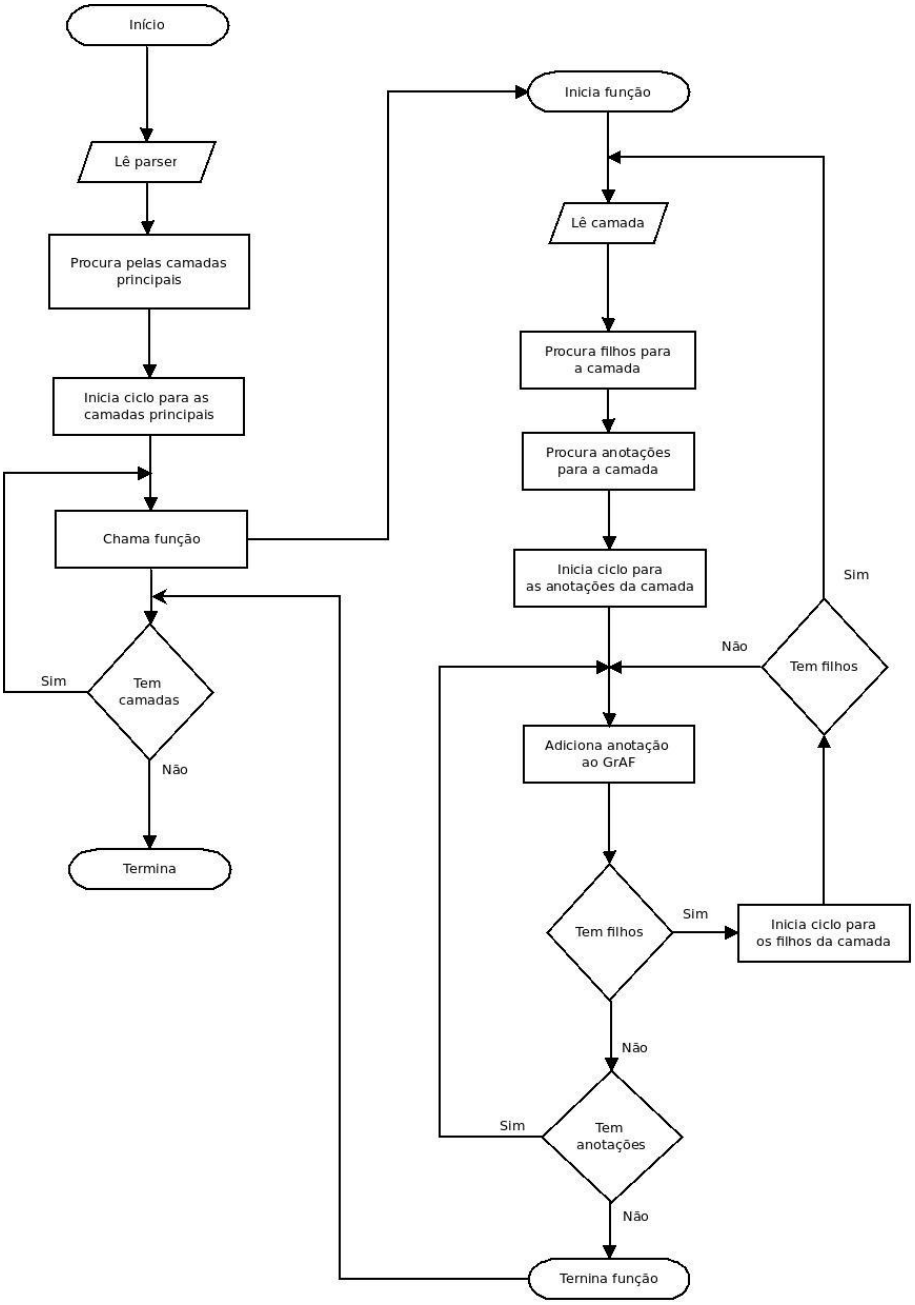

Figura 16 - Fluxograma do funcionamento do algoritmo GrAF Converter

Através deste estudo, foi possível concluir que não devem existir nós sem anotações. Para cada nó (*node*) pode apenas haver uma anotação (*annotation*), ter ou não uma região (*region*) e caso seja um nó descendente de outro, terá que ter um limite (*edge*) obrigatoriamente, conseguindo assim ter a vantagem de manter o controlo de quem é o nó pai (*parent node*).

Deste modo, desenvolveu-se um módulo totalmente transparente para o utilizador, responsável por transformar, criar e tratar um objeto de dados único num objeto GrAF, ao qual se deu o nome de "graf" na *package poioapi.io*.

Para que fosse possível garantir a objetividade dos dados, foi criada em primeiro lugar uma representação através de classes para:

- *Tier* (camada) – cada objeto *Tier* é composto por um nome (*name*) que é único e por uma *annotation_space*, que é uma etiqueta ou especificação para um determinado tipo de *Tier*;
- *Annotation* (anotação) – representação de uma anotação que tem um *id* único, um valor (*value*) e um conjunto de características (*features*);
- *PrimaryData* (dados primários) – representação dos dados primários, mais propriamente da origem dos dados usados para o valor das anotações, composto por um tipo (*type*), uma ligação externa (*external_link*), um nome de ficheiro (*filename*) e por conteúdo (*content*);
- *NodeId* – representação do *id* de um nó, de um limite (*edge*) e de uma região (*region*).

Posteriormente, foram criadas duas classes abstratas, *Abstract Base Classe* (ABC) [41] em *Python.* Uma delas, denominada *BaseParser*, encontra-se no módulo *poioapi.io.graf*, para ser utilizada como API dentro da biblioteca e criar a ligação entre os *parsers* (leitores) dos outros formatos de dados e a representação interna, apresentando os seguintes métodos:

- *get_root_tiers( )* – obtém todas as camadas principais (*root tiers*);
- *get_child_tiers_for_tier(tier)* – obtém as camadas filho (*child tiers*) de uma determinada camada;
- *get_annotations_for_tier(tier, annotation_parent)* – obtém todas as anotações para uma determinada camada e com uma anotação pai específica;
- *tier_has_regions(tier)* – verifica se uma anotação tem regiões;
- *region_for_annotation(annotation)* – obtém as regiões para uma determinada anotação;
- *get_primary_data( )* – obtém a fonte de dados das anotações.

A outra classe, com o nome *BaseWriter*, serve como modelo base para as classes criarem ficheiros a partir de objetos GrAF, tendo apenas o método *write(outputfile, graf_graph, tier_hierarchies, primary_data, meta_information).*

Após o desenvolvimento das classes descritas em epígrafe, foram criadas, por um lado, a *GrAFConverter*, que se trata de uma classe capaz de realizar o algoritmo de conversão descrito no início desta secção, tendo como responsabilidade fazer a conversão dos vários formatos de ficheiros e dados para um objeto GrAF. Por outro lado, a classe *Writer*, que é responsável pela escrita de um objeto GrAF em ficheiros.

3.3.2.3 Aplicabilidade do módulo GrAF

Após a descrição das classes presentes no módulo *poioapi.io.graf*, proceder-se-á à análise pormenorizada do funcionamento e do comportamento da classe *GrAFConverter*, aquando da sua utilização na passagem de um formato de ficheiros ou de dados para um objeto GrAF. Primeiramente, serão expostos as variáveis e os métodos importantes, bem como alguns conceitos utilizados, no sentido de possibilitar uma melhor compreensão do seu funcionamento. A figura 17 demonstra o aspeto interno da classe.

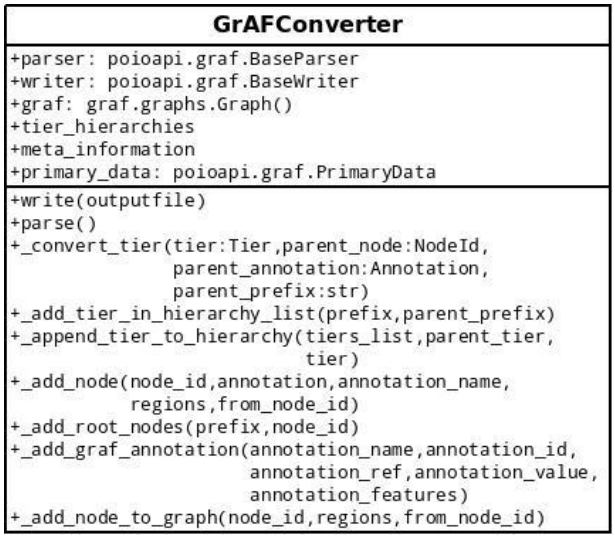

Figura 17 - Diagrama de classe GrAFConverter

A classe *GrAFConverter* é constituída pelas seguintes variáveis:

- *parser* – objeto do tipo *BaseParser*;
- *writer* – objeto do tipo *BaseWriter*;

- *graf* – objeto do tipo *Graph* da biblioteca *graf-python*, proveniente da classe *Graph* do módulo *graphs*, composto por uma lista de estruturas de caraterísticas (*FeatureStructure()*), limites (*GraphEdges()*), regiões (*IdDict()*), nós (*GraphNodes()*), um cabeçalho (*GraphHeader()*) e espaços de anotações (*GraphASpaces()*);
- *tier_hierarchies* – lista de *Data Strucutre Type* proveniente do ficheiro que está a ser processado. O exemplo seguinte mostra como pode ser a representação de uma *tier_hierarchies*:

```
[
      ['utterance..K-Spch'],

      ['utterance..W-Spch',
                  ['words..W-Words',
                        ['part_of_speech..W-POS']
                  ],
                  ['phonetic_transcription..W-IPA']
      ],

      ['gestures..W-RGU',
                  ['gesture_phases..W-RGph',
                        ['gesture_meaning..W-RGMe']
                  ]
      ],

      ['gestures..K-RGU',
                  ['gesture_phases..K-RGph',
                        ['gesture_meaning..K-RGMe']
                  ]
      ]
]
```

- *meta_information* – objeto do tipo *ElementTree* que contém informação considerada relevante sobre o ficheiro de origem das anotações dos dados primários mas que não pode ser armazenada em nenhum dos documentos do LAF. Esta variável pode servir também para fazer a reconstrução do ficheiro original, como é possível verificar no seguinte exemplo escrito:

```
<?xml version="1.0" encoding="UTF-8"?>
<ANNOTATION_DOCUMENT>
      <HEADER MEDIA_FILE="" TIME_UNITS="milliseconds">
                  <MEDIA_DESCRIPTOR MEDIA_URL="elan1.mpg"
MIME_TYPE="video/mpeg"/>
      </HEADER>
      <TIME_ORDER>
                  <TIME_SLOT TIME_SLOT_ID="ts1" TIME_VALUE="0"/>
            <TIME_SLOT TIME_SLOT_ID="ts2" TIME_VALUE="280"/>
            <TIME_SLOT TIME_SLOT_ID="ts3" TIME_VALUE="440"/>
      </TIME_ORDER>
    <LINGUISTIC_TYPE GRAPHIC_REFERENCES="false"
LINGUISTIC_TYPE_ID="utterance" TIME_ALIGNABLE="true"/>
</ANNOTATION_DOCUMENT>
```

- *primary_data* – objeto do tipo *PrimaryData* que contém a informação relativa aos atributos *loc* – caminho relativo ou nome do ficheiro dos dados primários – e *f.id* – tipo de ficheiro.

A classe enunciada é constituída por métodos principais e auxiliares. Relativamente aos métodos principais, consideram-se os seguintes:

- *parse( )* – método responsável por transformar um objeto *parser* no formato GrAF;
- *write(outputfile)* – método responsável por escrever um objeto GrAF em ficheiros GrAF/XML;
- *_convert_tier(tier, parent_node, parent_annotation, parent_prefix)* – método responsável por procurar os filhos e as anotações de cada camada (*Tier*) para depois as associar a um nó de cada anotação e posteriormente associar o nó ao objeto do grafo.

Por sua vez, os métodos auxiliares são:

- *_add_tier_in_hierarchy_list(prefix, parent_prefix)* – adiciona a uma lista de paridades entre as camadas pai filho – *_tiers_parent_list* –, através dos parâmetros *prefix* e *parent_prefix*;
- *_append_tier_to_hierarchy(tiers_list, parent_tier, tier)* – método recursivo para fazer uma pesquisa em profundidade por elementos do tipo lista numa lista de camadas;
- *_add_node(node_id, annotation, annotation_name, regions, from_node_id)* – contém os métodos para adicionar um nó e a sua anotação ao objeto *graf*;
- *_add_root_nodes(prefix, node_id)* – adiciona os nós principais (*root nodes*) ao cabeçalho do objeto *graf*;

- *_add_graf_annotation(annotation_name, annotation_id, annotation_ref, annotation_ _value, annotation_features)* – adiciona a anotação de um determinado nó ao objeto *graf*;
- *_add_node_to_graph(node_id, regions, from_node_id)* – adiciona um determinado nó ao objeto *graf*.

3.3.2.4 Funcionamento do Parsing

Para melhor compreender o funcionamento do *parsing*, é necessário, numa primeira abordagem, ter em conta a constituição e os conceitos dos objetos que compõem a variável *graf*:

- Uma anotação (*annotation*) é composta por um *annotation_name*, que corresponde ao tipo de anotação a que esta pertence – que normalmente é o mesmo valor do nome da camada quando esta não tem *annotation_space* que, no caso de existir, assume esse valor –; um *annotation_id*, que é a identificação única da anotação; um *annotation_ref*, o qual é o identificador ao qual o nó pertence; um *annotation_value*, o valor da anotação; por fim, um *annotation_features*, um dicionário com mais características para além do *annotation_value*.
- Um nó (*node*) é composto por um *node_id*. A composição do *id* do nó é obtida a partir da classe *NodeId*, a qual é composta por um prefixo e um índice (*index*). Este prefixo e sufixo são gerados automaticamente durante o *parsing* de um formato de dados. Geralmente o prefixo é igual ao nome da camada (*tier*). No entanto, quando as camadas (*tiers*) pertencem a um *annotation_space*, especificação ou tipo de anotação, o prefixo é formado pelo nome da anotação (*annotation_name*) juntamente com o nome da *tier*, ao qual o nó pertence, separada por "..", como por exemplo, "gestures..K-RGU". O índice é sempre igual ao *id* da anotação, precedida por um "n", de *node*, por exemplo "na1".
- Uma região (*region*) é composta por um conjunto de âncoras (*anchors*) ou um intervalo de valores, em que as unidades dependem do formato de dados, podendo ser por exemplo milissegundos, *tokens*, entre outros, e por um *id*, que por norma é igual ao *id* do nó a que corresponde mas onde a letra "n" no índice é substituída por um "r", de *region*.
- Por fim, *annotation_space* é uma lista com o agrupamento dos tipos de anotações.

A sequência lógica do *GrAFConverter* começa pelo *parsing* (leitura) de um determinado formato de dados, depois é efetuada uma conversão onde se vai posteriormente construir um objeto GrAF e armazená-lo na propriedade *graf*, como é exemplificado na figura 18.

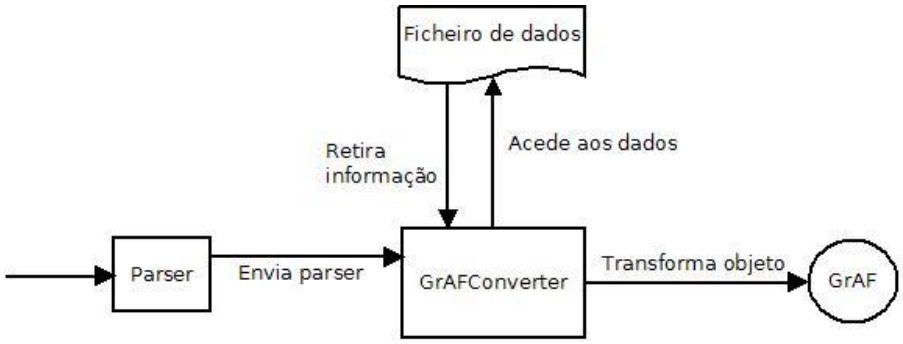

Figura 18 - Esquema básico de um parsing

O funcionamento da transformação de dados começa com o método *parse(),* que em primeiro lugar faz uma pesquisa das camadas principais (*root tiers*). Por cada camada principal que encontrar é efetuada uma chamada ao método *_convert()*, que no início procura pelos filhos desta camada e armazena-os numa lista, *child_tiers*. Este método vai ainda pesquisar o prefixo para os nós, o *annotation_name*, para as anotações da camada em causa e saber se esta tem regiões (*regions*). Depois faz uma recolha das anotações da camada, pesquisando apenas pelas anotações que tenham a mesma anotação pai (*annotation_parent*), terminando com o armazenamento do resultado na lista *annotations*. Quando encontradas anotações, é iniciado um ciclo em que para cada anotação é realizada a verificação de regiões, posteriormente cria um nó e, consequentemente, adiciona-o ao grafo. Por fim, é verificada a existência dos filhos da camada e, no caso de existirem, é feita uma chamada recursiva ao método *_convert()* que efetua todo este processo novamente. Para concluir o *parsing*, o método *parse()* processa a lista hierárquica de camadas e são obtidas as informações dos dados primários (*primary_data*) e da meta-informação (*meta_information*). A figura 19 demonstra o exposto, recorrendo a um esquema de funcionamento da transformação dos dados.

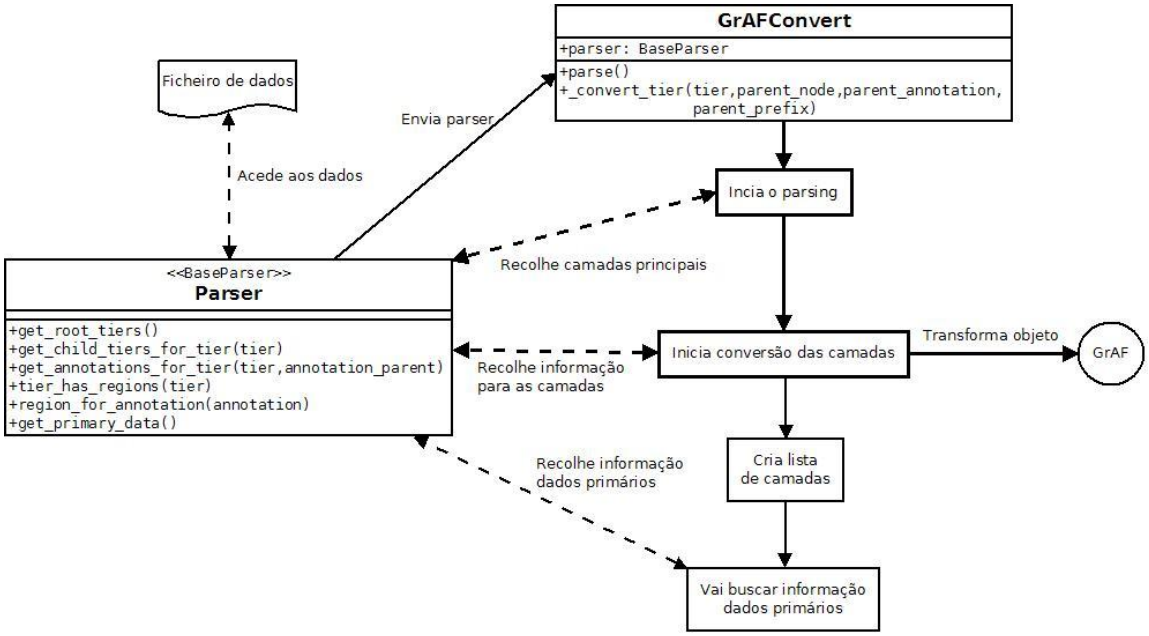

Figura 19 - Esquema de parsing

Em suma, a estrutura interna da representação GrAF no Poio API consiste fundamentalmente em três objetos, dos quais um tem uma lista de *Data Structure Types* ou estruturas hierárquicas de camadas; outro apresenta a origem dos dados das anotações; e finalmente, um objeto que é do tipo *Graph Annotation* oriundo da biblioteca *graf-python.* No apêndice A é demonstrada a aplicação de um simples *parser*.

3.3.2.5 Writing de Ficheiros GrAF/XML

Para fazer a escrita de ficheiros GrAF/XML foi criada a classe *Writer*, baseada na classe abstrata *BaseWriter*, que é constituída por um método considerado principal, *write()*.

Para além disso, esta classe é constituída por três variáveis, sendo elas *tier_hierarchies*, que é uma lista de *Data Structure Types*; *meta_information*, objeto que contém informação considerada relevante sobre o ficheiro de origem das anotações dos dados primários; *standoffheader*, objeto do tipo *StandoffHeader* da biblioteca *graf-python*. A figura 20 demonstra o aspeto interno da classe.

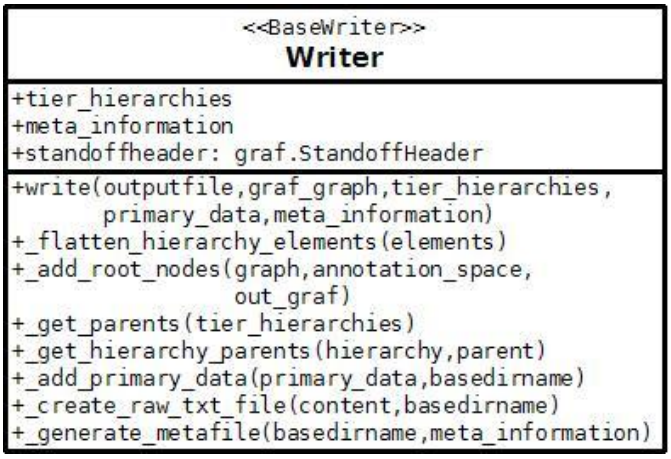

Figura 20 - Diagrama de classe Writer

Existem ainda sete métodos auxiliares para complementar o método *writer()*:

- *_add_root_nodes(graph, annotation_space, out_graf)* – adiciona os nós principais (*root nodes*) de um determinado espaço de anotação (*annotation space*) a um grafo;
- *_get_parents(tier_hierarchies)* – procura pelos pais de uma lista de camadas, ou seja, o elemento do nível mais alto;
- *_get_hierarchy_parents(hierarchy, parent)* – é auxiliar do método *_get_parents*;
- *_add_primary_data(primary_data, basedirname)* – verifica o tipo de dados primários e adiciona-os ao objeto *standoffheader*;
- *_create_raw_txt_file(content, basedirname)* – cria um ficheiro texto de um conteúdo oriundo do atributo da variável dos dados primários (*primary_data*);
- *_generate_metafile(basedirname, meta_information)* – gera um *metafile* do objeto *meta_information*.

O nome e os conteúdos dos ficheiros GrAF/XML dos documentos de anotação e do cabeçalho do documento dos dados primários são feitos com base nas suas estruturas hierárquicas ou *Data Structure Types*, isto é, cada objeto GrAF na biblioteca Poio API é acompanhado da variável *tier_hierarchies* que contém as *Data Structure Types* obtidas no processo de *parsing*. É a partir destas estruturas que será feita a escolha dos dados correspondentes a cada camada para um novo grafo, para posteriormente ser concretizada a sua renderização.

No sentido de respeitar as especificações do LAF, foram idealizadas duas soluções, das quais uma se destina à construção do nome dos ficheiros GrAF/XML e outra se refere ao tipo de dados primários. Deste modo, o nome dos ficheiros será baseado no valor do parâmetro de entrada *outputfile*, que, na maioria dos casos, deverá coincidir com o dos dados primários, seguido por um "-" e depois pelo nome da camada com a extensão de ficheiro ".xml". O ficheiro de cabeçalho dos dados primários tem o nome igual ao do parâmetro de entrada, mas com a extensão de ficheiro ".hdr". Quanto aos tipos de dados, foi decidido que existiriam quatro tipos fixos: *AUDIO* para áudio; *VIDEO* para vídeo; *TEXT* para texto; *NONE* para dados que não correspondem aos tipos enunciados previamente, bem como para situações em que se desconhece o tipo de dados.

O processo de escrita é feito pelo método *write()* que primeiramente procura os pais das *Data Structure Types* na lista de hierarquias (*tier_hierarchies*), para posteriormente adicionar ao cabeçalho a dependência da camada que está a ser escrita. Por conseguinte, é efetuado um nivelamento de todos os elementos na lista de camadas hierárquicas, através do método auxiliar *_flatten_hierarchy_elements* e é gerado um objeto *standoffrenderer* – do tipo *StandoffHeaderRenderer()* biblioteca *graf-python*. Quando terminado o nivelamento, é iniciado um ciclo para percorrer os elementos resultantes. No decorrer do ciclo é criado um objeto grafo limpo do tipo *Graph()* da biblioteca *graf-python* e um objeto *renderer* para fazer a renderização deste novo grafo. Ainda no ciclo, para cada camada, são filtrados os nós, limites, regiões, nós principais, espaços de anotação e o cabeçalho, para que possam ser adicionados ao novo grafo. Ao terminar o ciclo é adicionada a informação relacionada com os dados primários no cabeçalho do documento dos dados primários. Por fim, é realizada a renderização do cabeçalho do documento dos dados primários e, no caso de existir *meta_information*, é formado um ficheiro XML com essa informação. A figura 21 tem como objetivo demonstrar o processo descrito anteriormente. A legenda dos símbolos é apresentada no apêndice D.

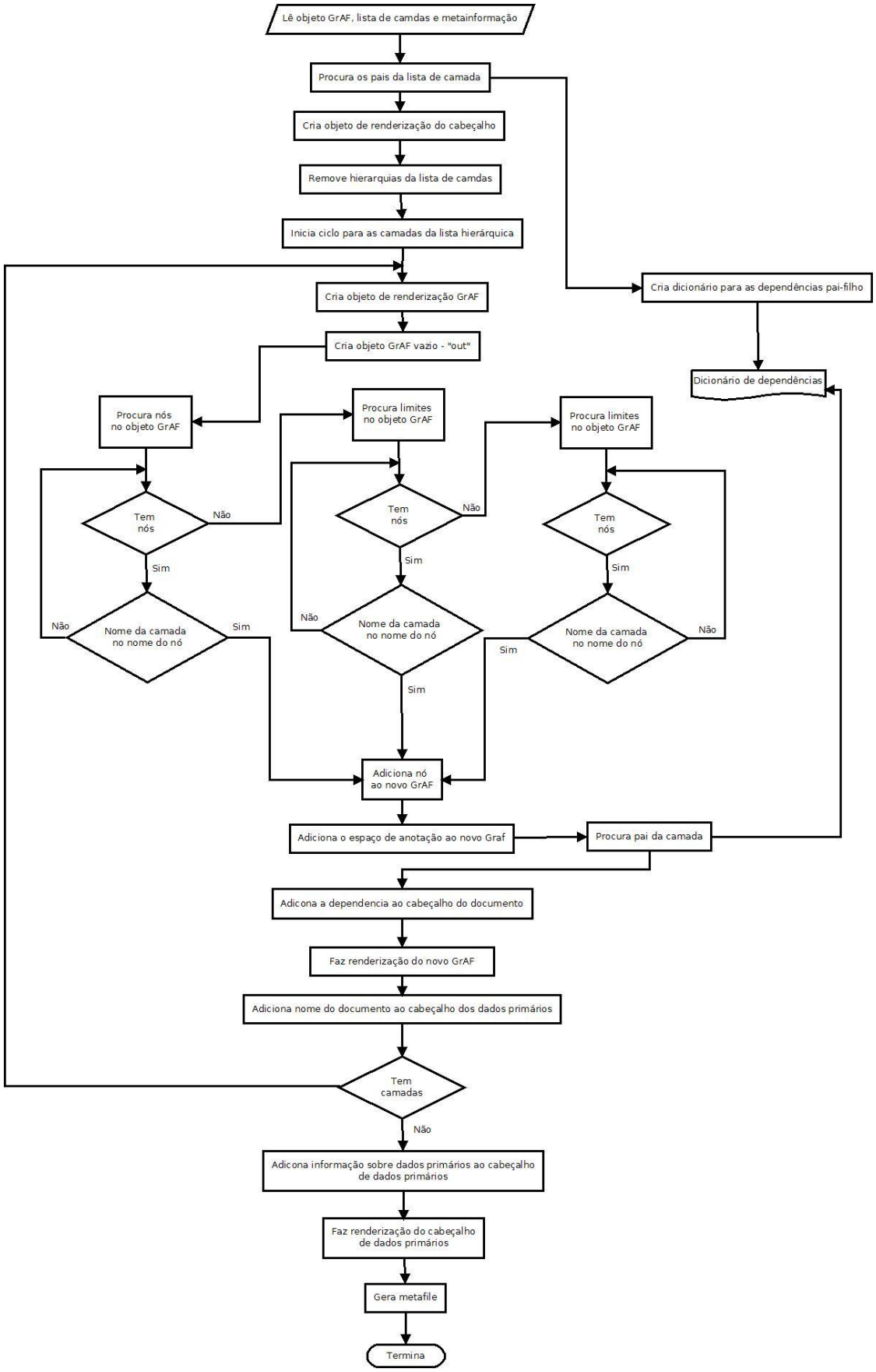

Figura 21 - Esquema de escrita de ficheiros GrAF/XML

3.3.2.6 Pesquisas nos grafos de anotação

Para possibilitar as pesquisas nos grafos de anotação na biblioteca Poio API, foi criada uma classe *AnnotationGraphFilter* no módulo *annotationgraph*, sendo que a sua utilização só pode ser feita juntamente com a classe *AnnotationGraph* – responsável por armazenar a informação relativa à representação do GrAF, como grafo de anotações – também do mesmo módulo. O objetivo desta classe é atribuir conjuntos de filtros aos grafos de anotações para que seja feita uma redução do seu tamanho em subconjuntos de grafos de anotação.

Este agrupamento de filtros ou lista é denominado de *filter chain* (filtro em cadeia). Cada filtro é composto por uma unidade de condições de pesquisa – que podem ser simplesmente *strings* ou expressões regulares – para cada uma das camadas previamente obtidas a partir de um *parsing*, que neste caso serão elementos do tipo *Data Structure Types*. No apêndice B é apresentada uma implementação dos filtros.

3.4 Metodologias de Desenvolvimento

Durante o desenvolvimento da biblioteca Poio API e das modificações da biblioteca *graf-python* foram imputados alguns princípios da metodologia *Agile* [30] e *Test-driven Development* (TDD) [31] e utilizado o processo de desenvolvimento *Kanban* [32]. A escolha deste processo resultou essencialmente do facto de este projeto ser um *software* de carácter científico e de estar a ser acompanhado e utilizado durante o seu desenvolvimento, o que obrigava que o *workflow* não fosse invariável, ou seja, era necessário existir a possibilidade de trocar e dar outra prioridade às tarefas entre os estados de evolução. Todo o acompanhamento de tarefas foi feito com o recurso a uma ferramenta online de gestão de projetos *Kanban* com o nome de *Kanbanize*[34].

Os testes desenvolvidos durante a conceção de ambas as bibliotecas foram sempre realizados com a tecnologia *NoseTests*. Os *NoseTests* são uma coleção de subclasses do módulo *unittest.TestCase* para a criação de testes na linguagem de programação *Python* [33]. O *unnittest – Unit Testing Framework* é uma *framework* para testes baseada em

[34] http://kanbanize.com/, acedido 09/09/2013

JUnit, que garante quatro conceitos importantes: “test fixture”, consiste na representação da preparação necessária para realizar um ou mais testes; “test case“, unidade individual de testes; “test suite”, coleção de “test cases”, “test suites” ou de ambos; “test runner”, componente que realiza a execução dos testes e fornece um resultado ao utilizador [34].

Os conjuntos de testes criados neste desenvolvimento das bibliotecas Poio API e *graf-pyhton* foram essencialmente realizados para verificar o funcionamento dos *parsers* criados. Todos os testes seguiram as normas da tecnologia dos *NoseTests*, que implica a existência dos sufixos “test” e “Test” antes do nome dos métodos e classes de modo a serem executados. No apêndice C é possível verificar um exemplo da implementação do teste sobre o *parser* para o formato de dados EAF, do *software* ELAN.

Casos de Uso

Apresentada a proposta para o projeto de *software* no capítulo 3, importa agora demonstrar a sua aplicabilidade num contexto real e comparar a sua potencialidade em relação ao problema exposto. Deste modo, este capítulo tem como intuito apresentar alguns dos casos de sucessos que foram conseguidos durante e após o desenvolvimento da solução e do *software*.

4.1 Parser

Parser é um leitor de texto e vai ser através deste que se vai conseguir extrair a informação dos vários ficheiros existentes e utilizados na prática da linguística. O Poio API teve sempre como princípio abstrair o utilizador da representação GrAF, ou seja, a única preocupação centra-se na extração da informação, consoante o formato desta para ser utilizada pela biblioteca, afim da criação de um objeto GrAF, como se pode verificar na figura 22.

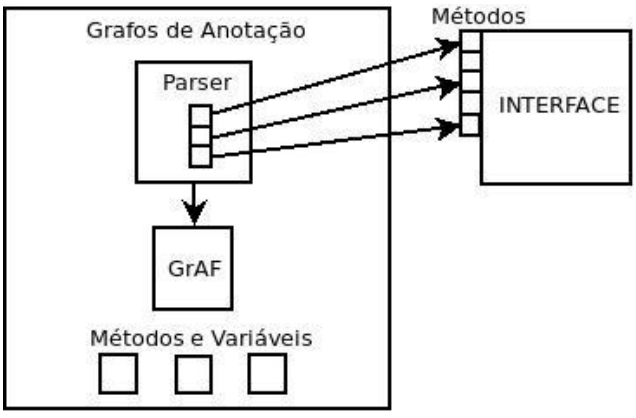

Figura 22 - Representação abstrata Poio API

O principal objetivo deste projeto era conseguir transformar ficheiros no formato *Elan Annotation Format* (EAF) no modelo de dados unificado, LAF. No entanto, foram desenvolvidos mais cinco *parsers* que, de certa forma, vieram reforçar e provar o funcionamento desta biblioteca, TypeCraft, *Text Corpus Format* (TCF), Toolbox,

Spreadsheet e *pickle*. De seguida, irão ser introduzidos os programas e a descrição dos seus formatos de dados dos ficheiros, para os quais foram desenvolvidos os estudos e a solução para o objeto *parser* formado por cada um deles. Este objeto *parser* consiste numa classe baseada na classe abstrata *BaseParser*, da biblioteca Poio API, dentro do módulo criado para cada um dos tipos de ficheiros enunciados anteriormente.

4.1.1 ELAN

ELAN (*EUDICO*[35] *Linguistic Annotator*) é uma ferramenta que permite a criação, edição e pesquisa de anotações sobre os recursos de vídeo e áudio. Foi desenvolvida no Instituto Max Planck em *Nijmegen*, Holanda, com o objetivo de explorar as anotações em dados multimédia. Embora tenha sido concebido essencialmente para o estudo e análise de línguas, linguagem gestual e movimento, pode ser utilizado para outros fins, como por exemplo, documentação de línguas ameaçadas [35] [36].

A utilização do ELAN faz com que o utilizador tenha que ter em consideração, para além das *Tier* (camadas) e das *Annotations* (anotações), o conceito *Type* (tipo) ou *Linguistic Type* (tipo linguístico).

Cada camada é um conjunto de anotações que partilham as mesmas características, estando associada a um determinado tipo linguístico (*Linguistic Type*), que consiste num número de restrições que se podem aplicar numa camada e, por sua vez, nas suas anotações e também num tipo de dados linguísticos, como tradução, ortografia, morfossintático (*part-of-speech*), entre outros. Existem dois tipos: Camadas Independentes, que contêm anotações diretamente ligadas a um intervalo de tempo, conhecidas como "time-alignable type"; e Camadas de Referência, também conhecidas como "symbolic", que contêm as anotações que estão ligadas a anotações de outras camadas que geralmente não estão ligadas diretamente ao eixo do tempo, no entanto, podem pertencer a um intervalo de tempo determinado pela *Parent Tier* (camada pai) [37] [38].

[35] http://www.mpi.nl/tg/lapp/eudico/eudico.html, acedido 24/08/2013

Um ficheiro EAF tem uma estrutura semelhante a um ficheiro XML e tem que ter sempre presente pelo menos os seguintes elementos [23]:

- *ANNOTATION_DOCUMENT* – elemento principal do documento;
- *HEADER* – este elemento contém a informação sobre o tipo de ficheiros media, como o nome do ficheiro e a unidade de tempo;
- *TIME_ORDER* – elemento que contém os intervalos de tempos das anotações, *TIME_SLOT*, ordenados. As *TIME_SLOT* indicam um ponto único na linha de tempo de um media, podendo, em muitos casos, ser consideradas como âncoras. Têm dois atributos *TIME_SLOT_ID* e *TIME_VALUE*. No entanto, o *TIME_VALUE* é opcional, o que permite que uma *TIME_SLOT* possa servir como âncora ou como um marcador numa sequência parcial de anotações alinhadas. É ainda possível existirem várias *TIME_SLOTs* com o mesmo *TIME_VALUE*;
- *TIER* – representação de uma camada, contendo os elementos das anotações. As anotações são representadas pelo elemento *ANNOTATION* que, consoante o seu tipo linguístico, podem ser dois tipos de anotação: *ALIGNABLE_ANNOTATION,* utilizada para as camadas que sejam do tipo *time alignable*, ou seja, independentes com anotações associadas a um segmento media, compostas sempre por um *id* e por dois *TIME_SLOTs*; e *REF_ANNOTATION*, destinada a camadas que contenham anotações que não estejam ligadas diretamente a um segmento media e indiquem valores para outras anotações, sendo composta por um *id* e por *ANNOTATION_REF*, referência da anotação para a qual aponta;
- *LINGUISTIC_TYPE* – elemento responsável por um tipo linguístico;
- *LOCALE* – identificador da língua utilizada nas anotações;
- *CONSTRAINT* – especifica as restrições aplicadas a uma camada e às suas anotações, através do elemento *LINGUISTIC_TYPE*;
- *CONTROLLED_VOCABULARY* – contentor de elementos CV_ENTRY, responsáveis por guardar as abreviaturas de palavras usadas nas anotações, como por exemplo, “noun” pode ser abreviado por “n”.

Seguidamente, é mostrado um exemplo escrito de como pode ser representado um ficheiro do formato EAF.

```
<?xml version="1.0" encoding="utf-8"?>
<ANNOTATION_DOCUMENT DATE="2013-09-09T09:09:09" AUTHOR="alopes"
VERSION="1" FORMAT="2.3">
  <HEADER MEDIA_FILE="" TIME_UNITS="milliseconds">
    <MEDIA_DESCRIPTOR MEDIA_URL="example1.mpg" MIME_TYPE="video/mpeg"/>
  </HEADER>
  <TIME_ORDER>
    <TIME_SLOT TIME_SLOT_ID="ts1" TIME_VALUE="0"/>
    <TIME_SLOT TIME_SLOT_ID="ts2" TIME_VALUE="10"/>
  </TIME_ORDER>
  <TIER TIER_ID="W-Words" PARTICIPANT=""
LINGUISTIC_TYPE_REF="word" DEFAULT_LOCALE="en">
    <ANNOTATION>
            <ALIGNABLE_ANNOTATION ANNOTATION_ID="a1"
TIME_SLOT_REF1="ts1" TIME_SLOT_REF2="ts2">
                <ANNOTATION_VALUE>so</ANNOTATION_VALUE>
            </ALIGNABLE_ANNOTATION>
        </ANNOTATION>
  </TIER>
  <TIER TIER_ID="W-POS" PARTICIPANT=""
LINGUISTIC_TYPE_REF="POS" DEFAULT_LOCALE="en" PARENT_REF="W-Words">
    <ANNOTATION>
            <REF_ANNOTATION ANNOTATION_ID="a2" ANNOTATION_REF="a1">
                <ANNOTATION_VALUE>v</ANNOTATION_VALUE>
            </REF_ANNOTATION>
        </ANNOTATION>
  </TIER>
  <LINGUISTIC_TYPE LINGUISTIC_TYPE_ID="utterance" TIME_ALIGNABLE="true"
GRAPHIC_REFERENCES="false"/>
  <LOCALE LANGUAGE_CODE="en" COUNTRY_CODE="US"/>
  <CONSTRAINT STEREOTYPE="Included_In" DESCRIPTION="Included"/>
  <CONTROLLED_VOCABULARY CV_ID="POS" DESCRIPTION="Part of Speech">
    <CV_ENTRY DESCRIPTION="verb">v</CV_ENTRY>
  </CONTROLLED_VOCABULARY>
</ANNOTATION_DOCUMENT>
```

O *parser* para o ficheiro de formatos EAF foi feito com base no *Linguistic Type*, o que significa que as estruturas hierárquicas de camadas que o Poio API formar vão ser sempre agrupadas pelas *Linguistic Type* e não pelas *Tiers*. Assim, foi necessário fazer uma subclasse da classe *Tier* do módulo *graf* do Poio API, com o nome *ElanTier*, na qual foi acrescentado o atributo de "linguistic_type". Resumidamente, o Poio API vai extrair todos os valores dos elementos *ANNOTATION* do ficheiro EAF e converter os valores em nós e anotações do formato GrAF. As regiões vão ser criadas com base nos elementos das *TIME_SLOTs* e a restante informação vai ser armazenada no ficheiro de metadados.

4.1.1.1 Funcionamento e resultados do *parsing*

Existe um método auxiliar, *_parse( )*, que em primeiro lugar vai encontrar o elemento principal do ficheiro (*ANNOTATION_DOCUMENT*), o qual vai servir de suporte para todas as pesquisas feitas pelo *parser*. Posteriormente, é executado outro método auxiliar que vai mapear os elementos

TIME_SLOTS do elemento *TIME_ORDER* para um dicionário, de modo a assegurar uma maior rapidez na pesquisa dos valores, uma vez que os intervalos de tempo das anotações são definidas apenas com os *ids* das *TIME_SLOTS* e não com os valores reais que elas representam. Este mapeamento tenta também descobrir os valores em falta, uma vez que o *software* ELAN permite que existam *TIME_SLOTS* sem valor (*TIME_VALUE*), para isto foi criada uma equação que calcula o valor em falta da seguinte forma: procurar pelas anotações que contenham o mesmo *TIME_SLOT_ID* numa das referências do intervalo – se o *id* estiver na primeira parte do intervalo guarda-se a segunda parte e vice-versa – seguidamente, é feito um somatório dos valores encontrados e por fim é feita a divisão do somatório pelo número de elementos encontrados.

$$time\_slot\_value = \sum range\_time\_slots \div len(range\_time\_slots)$$

Depois do mapeamento são passados os metadados para a variável *meta_information*. Os metadados são compostos por toda a informação do ficheiro, exceto as anotações das camadas, sendo esta preservada exatamente como está, incluindo os valores *TIME_SLOTs* atualizados, pelo que, desta maneira, será possível fazer uma reconstrução deste ficheiro.

Os restantes métodos oriundos da *BaseParser* trabalham da seguinte forma:

- *get_root_tiers* – vai procurar todas as camadas nos elementos “TIER” que não tenham o atributo “PARENT_REF”, responsável por especificar se uma determinada camada tem pai ou se é um pai;
- *get_child_tiers_for_tier* – procura por todas as camadas cujo atributo “PARENT_REF” seja igual ao nome do objeto *tier*.
- O método *get_annotations_for_tier* – vai diferenciar na pesquisa de anotações, uma vez que tem dois tipos de anotações. No caso de ser uma camada do tipo *time_alignable* que não tenha anotação pai (*annotation_parent*), devolve todas as anotações. Havendo anotação pai, procura pela anotação ou anotações cujos intervalos de tempo estejam dentro do intervalo definido na anotação pai. Não sendo uma camada do tipo *time_alignable*, a pesquisa é feita pela anotação que tenha o valor do atributo *ANNOTATION_REF* igual ao *id* do *annotation_parent* que é enviado com a camada;
- *tier_has_region* – devolve verdadeiro para todas as camadas cujo tipo linguístico seja *time_alignable*;
- *region_for_annotation* – utiliza o dicionário feito previamente para encontrar os valores

TIME_VALUE para cada uma das *TIME_SLOT_IDs* que a anotação tem associadas;

- *get_primary_data* – através do elemento *MEDIA_DESCRIPTOR* procura o tipo e a origem do tipo de dados primários. Com recurso ao atributo *MIME_TYPE* é possível verificar se o tipo de dados é vídeo ou áudio e com o atributo *MEDIA_URL* é possível verificar a sua ligação externa ou localização.

Através do exemplo escrito anteriormente no inicio da seção, e das figuras 23 e 24, é possível verificar o resultado do objeto *graf* no Poio API e dos ficheiros GrAF resultantes. Recorrendo à análise da figura 24, é possível constatar que os ficheiros GrAF gerados são respetivamente iguais aos nomes das estruturas de camadas hierárquicas obtidas durante o *parsing*. Para além dos ficheiros GrAF, é ainda gerado um ficheiro *metafile*.

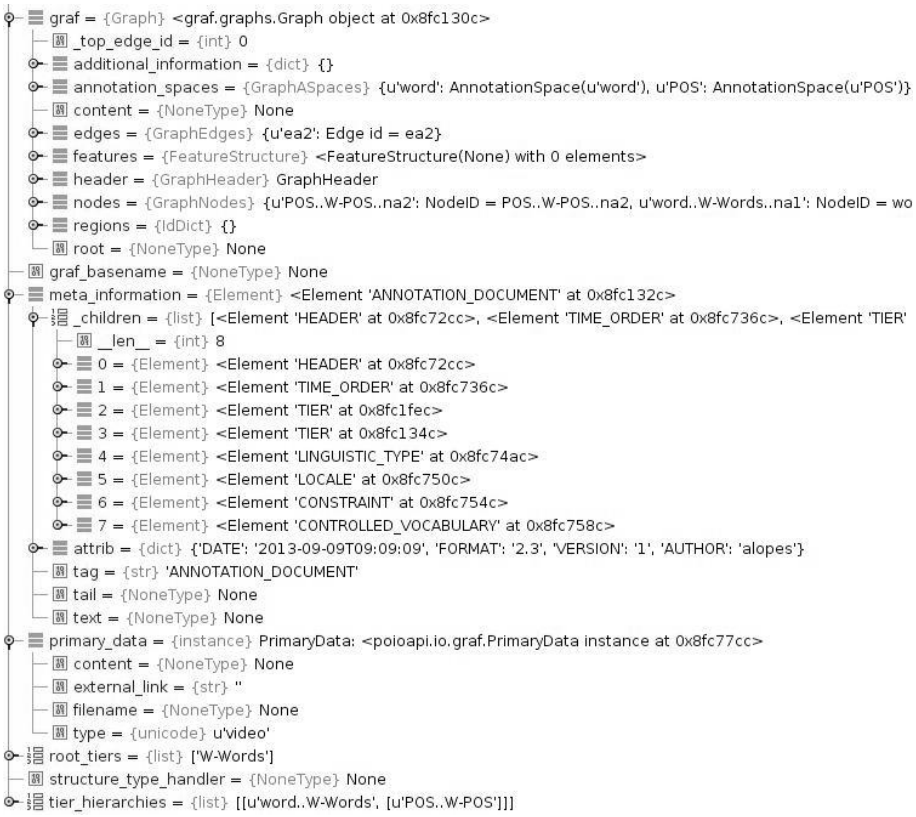

Figura 23 - Resultado de um objeto GrAF após parsing de um ficheiro EAF

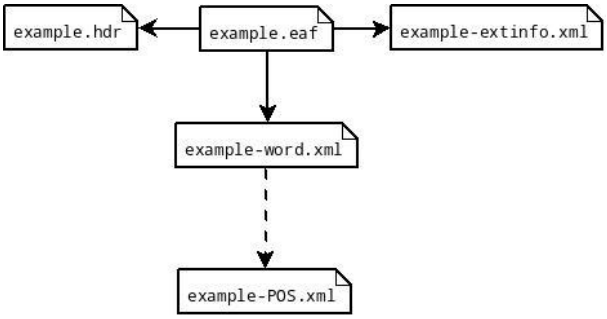

Figura 24 - Resultado dos ficheiros GrAF/XML após parsing de um ficheiro EAF

4.1.2 TypeCraft

TypeCraft é um editor de anotações online e uma base de dados online de linguagem natural, em que o conteúdo é constituído por textos multi-línguas anotados linguisticamente, isto é, os textos existentes na base de dados são importados ainda em *raw* (consiste nos dados que ainda não sofreram qualquer tipo de modificações) e depois podem ser anotados com as ferramentas embutidas no site. Este projeto tem a particularidade de permitir aos utilizadores criarem e partilharem projetos individuais com outros utilizadores ou colaboradores, podendo também torná-los públicos através de páginas *wiki*. Para além disso, existe também a possibilidade de exportar os textos já anotados para os formatos de *MS Word* (.doc), *OpenOffice.org (.odt)*, *LaTeX* e XML [29] [39].

Internamente, o *kernel* (núcleo de processamento) que o TypeCraft utiliza, trabalha apenas ao nível morfológico das palavras, o que significa que apresenta uma estrutura hierárquica de camadas fixa, como a seguinte [39]:

```
[ ' phrase ',
     [ ' word ', ' pos ',
          [ ' morpheme ' , ' gloss ' ]
     ],
  ' translation ' , ' description ' ]
```

Embora o projeto permita que os dados sejam exportados para vários formatos de dados, o *parser* criado apenas lê ficheiros com uma estrutura em XML como o exemplo seguinte:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<typecraft xsi:schemaLocation="http://typecraft.org/typecraft.xsd"
xmlns="http://typecraft.org/typecraft"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <phrase valid="VALID" id="1">
        <original>Mpa k?se? nk? dan no mu</original>
        <translation>The big bed cannot enter the room</translation>
        <description></description>
        <globaltags tagset="Default" id="1"/>
        <word head="false" text="Mpa">
            <pos>N</pos>
            <morpheme meaning="bed" baseform="mpa" text="mpa">
                <gloss>SBJ</gloss>
            </morpheme>
        </word>
        <word head="false" text="kɛseɛ">
            <pos>ADJ</pos>
            <morpheme meaning="big" baseform="kɛseɛ" text="kɛseɛ"/>
        </word>
      </phrase>
</typecraft>
```

4.1.2.1 Funcionamento e resultados do *parsing*

Primeiramente, com o recurso a um método auxiliar é criado um dicionário que vai conter toda a informação do ficheiro organizada. De acordo com o exemplo apresentado anteriormente, apenas os elementos <*phrase*> e <*globaltags*> contêm *id*, o que impossibilita que o rasto dos filhos e dos pais dos restantes elementos durante um *parsing* se mantenham. O dicionário referido armazena a informação sobre os pais e valores para cada elemento, assim como o seu "novo" *id*, que é gerado com variáveis incrementais para cada um dos elementos que não tem *id*, como por exemplo, *word*. Será neste dicionário que o *parsing* será feito, tornando também o processamento mais rápido.

Uma vez que a estrutura dos ficheiros é fixa, os filhos para as camadas também o são. Deste modo, os métodos implementados do *BaseParser* vão funcionar da seguinte forma:

- *get_root_tiers* – obtém a camada principal designada *phrase*;
- *get_child_tiers_for_tier* – permite saber que as camadas filhas da camada *phrase* são as camadas *word*, *translation* e *description*, sendo que para a camada *word*, as

filhas são *pos* e *morpheme*, pelo que para esta última a camada filha será *gloss*;

- *region_for_annotation* e *tier_has_regions* – existe uma particularidade para este *parser* que é o facto de não existirem regiões para as anotações deste formato, logo os métodos não devolvem qualquer informação;
- *get_annotations_for_tier* – faz uma procura no dicionário pelas anotações, cujo pai seja igual à anotação pai (*annotation_parent*);
- *get_primary_data* – foi deliberado que não existiriam dados primários para este formato e que a informação para o cabeçalho seria "unknown" relativamente ao nome do ficheiro e "NONE" para o tipo de dados primários.

Considerando o exemplo anterior e as figuras 25 e 26, é possível analisar os resultados obtidos durante o *parsing* descrito.

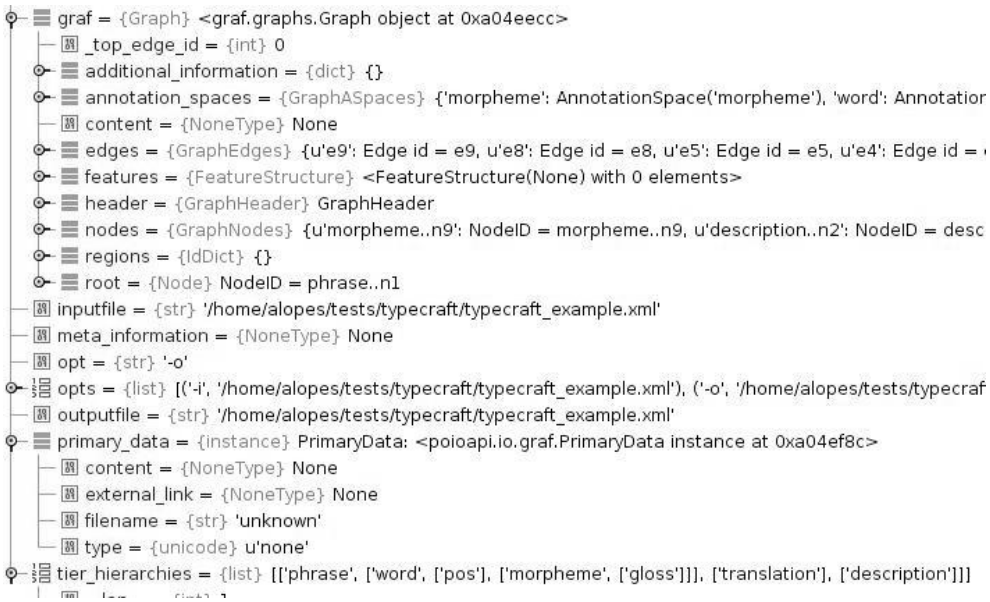

Figura 25 - Resultado de um objeto GrAF após parsing de um ficheiro TypeCraft

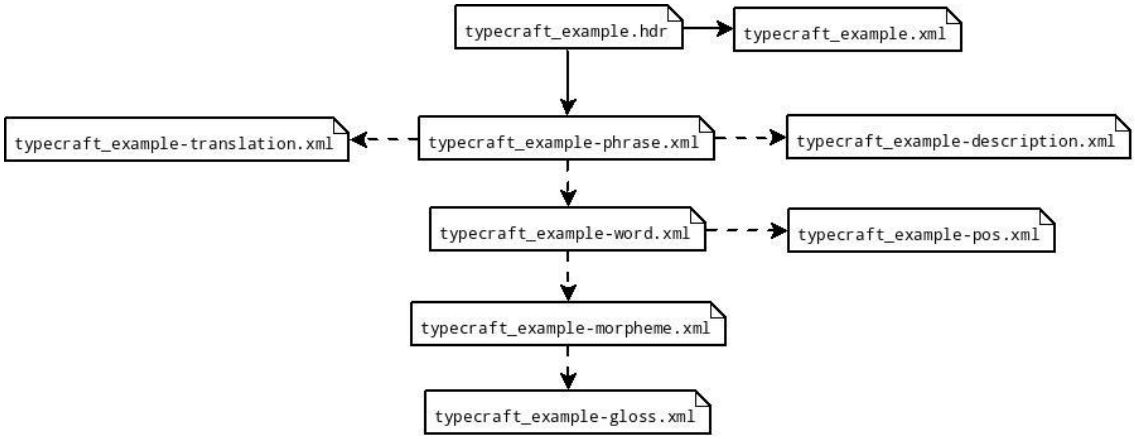

Figura 26 - Resultado dos ficheiros GrAF/XML após parsing de um ficheiro TypeCraft

4.1.3 Text Corpus Format (TCF)

WebLicht é um serviço de organização e execução de ambientes para anotação incremental automática de corpora, construído sobre os princípios de um *Service Oriented Architecture* (SOA) e é composto por um conjunto de serviços distribuídos para o processamento de dados; um repositório com metainformação sobre os serviços; e uma aplicação *web* com uma GUI, que permite a construção e execução de serviços relacionados [40].

Para além disso, é um formato de partilha de dados em XML que foi desenvolvido no âmbito da arquitetura *WebLicht*, no sentido de facilitar a eficiência de interoperabilidade entre ferramentas. Este formato permite que sejam aglomeradas várias anotações linguísticas, criadas por ferramentas do *WebLicht* num só ficheiro [27].

O formato TCF consegue atingir um grande volume de informação, visto que permite o agrupamento de vários tipos de anotações linguísticas e o armazenamento da informação sobre as ferramentas utilizadas para escrever as anotações, aspectos que podem ser verificados no exemplo abaixo.

```
<?xml version="1.0" encoding="UTF-8"?>
 <D-Spin xmlns="http://www.dspin.de/data" version="0.4">
    <MetaData xmlns="http://www.dspin.de/data/metadata">
        ….
    </MetaData>
    <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="pt">
        <text>Alé. Jordas cópia</text>
        <tokens>
            <token ID="t_0">Alé</token>
            <token ID="t_1">jordas</token>
            <token ID="t_2">cópia</token>
        </tokens>
        <sentences>
            <sentence ID="s_0" start="1" end="4" tokenIDs="t_0"/>
            <sentence ID="s_1" start="5" end="17" tokenIDs="t_1 t_2"/>
        </sentences>
        <POStags tagset="stts">
            <tag ID="pt_0" tokenIDs="t_0">N</tag>
            <tag ID="pt_1" tokenIDs="t_1">V</tag>
            <tag ID="pt_2" tokenIDs="t_2">N</tag>
        </POStags>
        ...
    </TextCorpus>
 </D-Spin>
```

No entanto, apenas foi feita a recolha das informações relativas às *tags <sentences>*, <*tokens>*, <*POStags>* e <*lemmas>*, que fazem parte do elemento *TextCorpus*. Isto faz com que a estrutura hierárquica de camadas seja fixa:

```
[ ' sentences ',
     [ ' tokens ',
          [ ' POStags ' , ' lemmas ' ]
     ],
]
```

4.1.3.1 Funcionamento e resultados do *parsing*

Neste formato de dados, as camadas também são fixas, o que significa que o método *get_root_tiers* devolve apenas a camada *sentences*. Por sua vez, o método *get_child_tiers_for_tier* devolve a camada *tokens* para a camada *sentences.* Por

conseguinte, as filhas da camada *tokens* são *POStags* e *lemmas*. As anotações são encontradas através da anotação pai (*annotation_parent*) e do método *get_annotations_for_tier*, exceto se for uma camada do tipo *sentences*. As regiões são obtidas a partir dos atributos *start* e *end* dos elementos *sentence*. Os dados primários serão considerados do tipo texto e o seu conteúdo será feito com base nos valores das *strings* do elemento *text*.

Aplicando o *parsing* ao exemplo anterior, o resultado será o exposto nas figuras 27 e 28. Existe a particularidade de que neste formato de dados não existe fonte externa de dados primários. Deste modo, para além dos ficheiros GrAF gerados é também criado um ficheiro texto (com a extensão ".txt"), a partir do conteúdo do atributo *content* da variável *primary_data* no objeto *graf*.

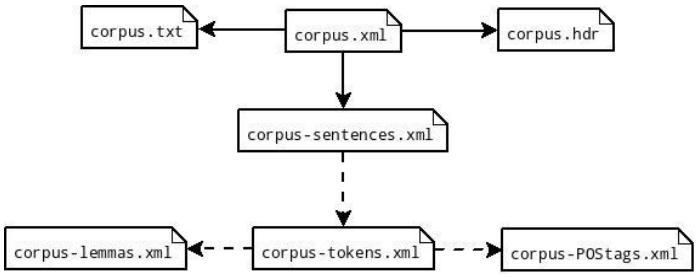

Figura 27 - Resultado dos ficheiros GrAF/XML após parsing de um ficheiro TCF

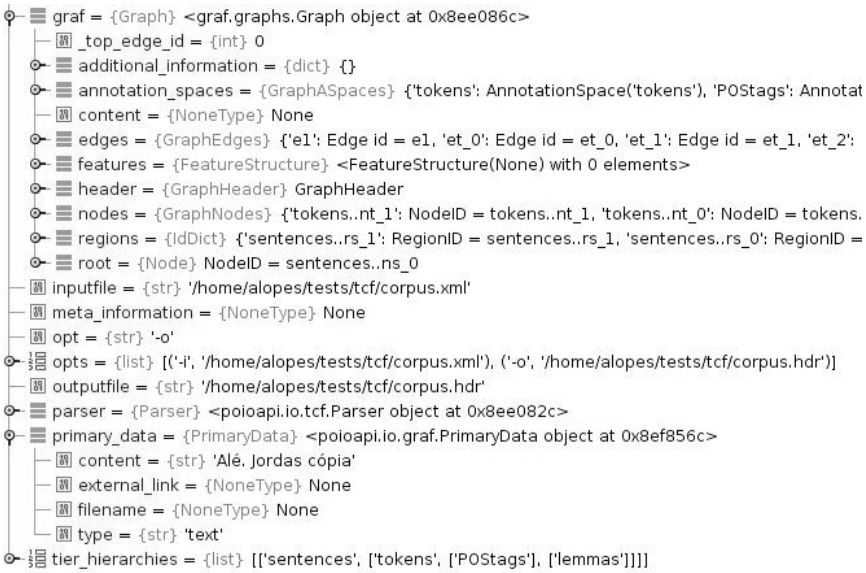

Figura 28 - Resultado de um objeto GrAF após parsing de um ficheiro TCF

4.2 Writer

Embora o objetivo principal fosse fazer uma transformação dos formatos de ficheiros para GrAF, o inverso também foi realizado com sucesso, o que permitiu que o *workflow* fosse alterado como é apresentado na figura 29. Um *writer* é aquele que escreve o objeto GrAF num formato de ficheiros específico, seguindo os requisitos mínimos obrigatórios. Com o Poio API foram desenvolvidos dois *writers*, um para o formato EAF e o outro para o formato *brat*, que é uma ferramenta *web-based* para fazer anotações de textos, preparada para trabalhar com anotações estruturadas, em que o conteúdo das notas não é de forma livre mas tem uma forma fixa que é automaticamente processada e interpretada por um computador [42]. No entanto, só irá ser abordado o assunto relativamente ao formato EAF.

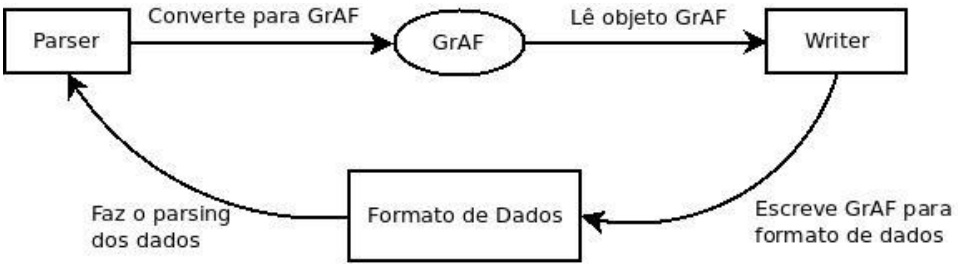

Figura 29 - Workflow do esquema parse/write da bibliteca Poio API

4.2.1 Transformação do GrAF em EAF

Para isto foi criado uma classe *Writer* proveniente da classe abstrata *BaseWriter* da biblioteca Poio API. Para que exista a possibilidade de fazer a transformação devem ser passadas por parâmetro do método *write( )* as seguintes variáveis: *graf_graph* – objeto com o grafo de anotações; *tier_hierarchies* – a lista hierárquica de camadas contém o nome das camadas que estão presentes nos elementos *<TIER>* do objeto *meta_information*; *primary_data* – objeto com a informação sobre os dados primários; por fim, *meta_information* – objeto que contém a informação sobre a estrutura do ficheiro EAF.

O funcionamento deste *writer* é realizado na totalidade pelo método *write( )* que primeiro faz a chamada a um método auxiliar que vai mapear os valores dos *TIME_SLOTs* para um

dicionário, de modo a que mais tarde seja possível fazer uma correspondência dos valores das regiões dos nós com os *TIME_SLOT_ID* de cada anotação. Depois de realizado um nivelamento à lista hierárquica de camadas, é iniciado um ciclo que vai percorrer todas as camadas. Este ciclo primeiramente tenta encontrar o elemento <*TIER*> na *meta_information* que tenha o mesmo *TIER_ID* que a camada atual e, no caso de ser positivo, procura no grafo de anotações os nós correspondentes à camada e, por conseguinte, pesquisa pelas anotações desses mesmos nós. Quando encontradas as anotações, é gerado um novo elemento <*ANNOTATION*> e adicionado ao elemento <*TIER*> em causa. No fim do ciclo é efetuada a escrita do ficheiro EAF, que antes faz uma atualização do elemento <*HEADER*> sobre dados primários.

Conclusão e Trabalho Futuro

O número de línguas ameaçadas e minoritárias têm vindo a aumentar nestes últimos 20 anos, no entanto, verifica-se, paralelamente a este aumento, o crescimento significativo do número de interessados e especialistas na preservação e revitalização destas línguas. Este grande interesse evidenciado faz com que as línguas não sejam extinguidas e causa o aumento do número de utilizadores de *softwares* de processamento linguístico e, por sua vez, o número de desenvolvimentos para novas estruturas e formatos de dados, possibilitando a evolução dos estudos de investigação nas diversas áreas da linguística.

Neste sentido, entende-se que os grafos GrAF podem ter um papel importante na implementação de *workflows* científicos na área da linguística, sendo determinantes para a dinâmica inerente ao processamento de linguagem.

O objetivo inicial do presente projeto consistia na obtenção de um processo que fizesse a transformação do formato de dados ELAN para o modelo de dados GrAF. No entanto, foi possível apurar a possibilidade de efetuar a transformação de outros tipos de formatos de dados e ficheiros. Deste modo, de acordo com o apresentado no capítulo Casos de Uso, alusivo à explicação e demonstração do funcionamento da biblioteca Poio API, existe a possibilidade de fazer um *workflow* com dois sentidos, isto é, de um formato de dados para GrAF e posteriormente, de GrAF para um novo formato, que no caso verificado foi o *Elan Annotation Format* (EAF). Esta consecução ajudou a provar a interoperabilidade que pode existir com a utilização do GrAF e dos vários formatos de ficheiros e, por conseguinte, as expectativas iniciais foram, de certa forma, superadas.

Neste sentido, considerando os objetos GrAF gerados pelo Poio API, os dados resultantes podem ser conduzidos para bibliotecas científicas em *Python*, tais como *networkx*, *numpy* ou *scipy*, de modo a possibilitar a comparação ou cruzamento de dados linguísticos ou outros, em projetos analíticos, como já é aplicado no projeto QuantHistLing[36].

[36] http://www.quanthistling.info/, acedido 21/09/2013

Foi ainda constatado que a conversão de formatos de ficheiros personalizados através do Poio API podem servir como um ponto de entrada para as estruturas oferecidas pela *American National Corpus* (ANC), e suportar a união de dados e anotações de múltiplas fontes de dados heterogéneos para análises posteriores.

Numa perspetiva futura, visto que ainda se denotam determinadas lacunas que poderão ser corrigidas, designadamente a nível de performance de alguns dos *parsers* existentes, pressupõe-se que o trabalho futuro centrar-se-ia na criação de um *parser* generalizado, capaz de fazer a leitura e transformação de um formato qualquer de dados e ficheiros para o modelo de dados GrAF. Para além disso, seria também importante explorar a possibilidade de fazer um processo inverso ao exposto anteriormente, ou seja, transformar objetos GrAF em formatos de dados e ficheiros aleatórios.

Bibliografia

[1] UNESCO Ad Hoc Expert Group on Endangered Languages. In International Expert *Meeting on UNESCO Programme Safeguarding of Endangered Languages*, Março 2003.

[2] WG 3: Linguistic Fieldwork, Anthropology and Language Typology, http://clarin-d.net/en/discipline-specific-working-groups/wg-3-linguistic-fieldwork-anthropology-language-typology.html, Agosto 2013.

[3] Jerome Feldman e George Lakoff, 2006, *Neural Theory of Language and Thought*, http://icbs.berkeley.edu/natural_theory_lt.php, Agosto 2013.

[4] Luc Steels e Joachim de Beule. A (very) Brief Introduction to Fluid Construction Grammar. In *Proceedings of the Third Workshop on Scalable Natural Language Understanding*, pp. 73 – 80, 2006.

[5] Nancy C. Chang e Benjamin K. Bergen. Embodied Construction Grammar in Simulation-Based Language Understanding. In *Construction Grammars: Cognitive Grounding and Theoretical Extensions*, pp. 147 – 190, 2005.

[6] Computational Issues in Fluid Construction Grammar, http://www.fcg-net.org/sample-page/computational-issues-in-fluid-construction-grammar, Março 2013.

[7] Blog oficial da Google, The Endangered Languages Project: Supporting language preservation through technology and collaboration, http://googleblog.blogspot.pt/2012/06/endangered-languages-project-supporting.html, Fevereiro 2013.

[8] Features Accentuate.us, http://accentuate.us/features.html, Fevereiro 2013.

[9] Kevin Scannell and Michael Schade, 2010. Accentuate Us!. In *Saint Louis University*, http://borel.slu.edu/pub/a.us.mathcs.pdf, Agosto 2013.

[10] Indigenous Blogs, http://indigenoustweets.com/blogs, Fevereiro 2013.

[11] Indigenous Tweets Blog, http://indigenoustweets.blogspot.pt/2011/03/welcomefailte.html, Fevereiro 2013.

[12] Nancy Ide , Laurent Romary and Eric de la Clergerie . International Standard for a Linguistic Annotation Framework. In *Journal Natural Language Engineering*, Vol. 10, Issue 3-4, pp. 211 - 225, Setembro 2004.

[13] Steven Bird and Mark Liberman. A formal framework for linguistic annotation. In *Speech Communication*, Vol. 33, Issues 1 – 2, pp. 1 – 2, 23 – 60, 2001.

[14] ISO 24612:2012: Language resource management – Linguistic annotation framework (LAF) International Organization for Standardization, http://www.iso.org/iso/catalogue_detail.htm?csnumber=63732, Geneva, Switzerland, Agosto 2012.

[15] Nancy Ide and Laurent Romary. Representing Linguistic Corpora and Their Annotations. In *Proceedings of the Fifth Language Resources and Evaluation Conference* (LREC), Genoa, Italy, 2006.

[16] Nancy Ide and Keith Suderman. GrAF: A graph-based format for linguistic annotations. In *Proceedings of the Linguistic Annotation Workshop*, pp. 1 – 8, Prague, Czech Republic, Junho 2007.

[17] Nancy Ide, Collin Baker, Christiane Fellbaum, Charles Fillmore and Rebecca Passonneau. MASC: A Community Resource For and By the People. In *Proceedings of Association for Computational Linguistics (ACL 2010)*, pp. 68 – 73, Uppsala, Sweden, Julho 2010.

[18] Nancy Ide and Keith Suderman. Bridging the gaps: interoperability for GrAF, GATE, and UIMA. In *Proceedings of the Third Linguistic Annotation Workshop*, pp. 27 – 34, Suntec, Singapore, Agosto 2009.

[19] Vera Ferreira, Peter Bouda and António Lopes. Poio API - An annotation framework to bridge Language Documentation and Natural Language Processing. In *Proceedings of the* Annotation of Corpora for Research in the Humanities (ACRH-2), Universidade Nova de Lisboa, 2012.

[20] The Language Archive, http://tla.mpi.nl/, Agosto 2013.

[21] The Language Archive - Language Data, http://tla.mpi.nl/home/language-data/, Agosto 2013.

[22] The Language Archive - Data Archive, http://tla.mpi.nl/resources/data-archive/, Agosto 2013.

[23] Elan Annotation Framework, http://www.mpi.nl/tools/elan/EAF_Annotation_Format.pdf, Agosto 2013.

[24] Marie Hinrichs, Thomas Zastrow and Erhard Hinrichs. WebLicht: Web-based LRT Services in a Distributed eScience Infrastructure. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation* (LREC'10), Valletta, Malta, May, 2010.

[25] Geoffrey Haig and Stefan Schnell, 2011. Annotations using GRAID, http://www.linguistik.uni-kiel.de/GRAIDmanuall, Agosto 2013.

[26] brat standoff format, http://brat.nlplab.org/standoff.html, Agosto 2013.

[27] The TCF Format, http://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/The_TCF_Format, Agosto 2013.

[28] WebLicht services with Servlet tutorial, http://weblicht.sfs.uni-tuebingen.de/WebLichtTutorial.pdf, Agosto 2013.

[29] TypeCraft, http://typecraft.org/tc2wiki/Main_Page, Agosto 2013.

[30] James Shore, *The Art of Agile Development*, O'Reilly Media Inc, 2008.

[31] Kent Beck, *Test-driven Development: By Example*, Addison-Wesley Professional, Novembro 2002.

[32] Henrik Kniberg and Mattias Skarin, *Kanban and Scrum: Making the Most of Both*, Lulu Enterprises Incorporated, 2010.

[33] NoseTests, http://nose.readthedocs.org/en/latest/index.html, Agosto 2013.

[34] Unit Tests Python, http://docs.python.org/3.3/library/unittest.html, Agosto 2013.

[35] ELAN Description, http://tla.mpi.nl/tools/tla-tools/elan/elan-description/, Agosto 2013.

[36] Linguistic Applications at the MPI, http://www.mpi.nl/world/tg/lapp/lapp.html, Agosto 2013.

[37] User Guide for ELAN Linguistic Annotator, http://www.mpi.nl/corpus/manuals/manual-elan_ug.pdf, Agosto 2013.

[38] ELAN - Linguistic Annotator Manual, http://www.mpi.nl/corpus/manuals/manual-elan.pdf, Agosto 2013.

[39] Scott Farrar. Review of TypeCraft from Pavel Mihaylov and Dorothee Beermann. In *Language Documentation & Conservation*, Vol. 4, pp. 60 – 60, 2010.

[40] WebLicht, http://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/Getting_Started, Setembro 2013.

[41] Abstract Base Classes, http://docs.python.org/2/library/abc.html, Agosto 2013.

[42] brat, http://brat.nlplab.org/introduction.html, Agosto 2013.

Apêndices

Do estudo feito durante esta dissertação resultou a publicação do artigo:
Vera Ferreira, Peter Bouda and António Lopes. 2012. "Poio API - An annotation framework to bridge Language Documentation and Natural Language Processing" apresentado no evento "Annotation of Corpora for Research in the Humanities" (ACRH-2) na Universidade Nova de Lisboa e pode ser lida em Mambrini, Francesco / Passarotti, Marco / Sporleder, Caroline (eds.) 2012. Proceedings of the Second Workshop on Annotation of Corpora for Research in the Humanities. Lisboa: Edições Colibri, 15-26.

A. Exemplo da implementação de um Parser

```
class SimpleParser(poioapi.io.graf.BaseParser):
      tiers = ["utterance", "word"]
      utterance_tier = ["This is a utterance", "that is another
utterance"]
      word_tier = [['This', 'is', 'a', 'utterance'], ['that', 'is',
'another',
      'utterance']]

      def __init__(self):
            pass

      def get_root_tiers(self):
            return [poioapi.io.graf.Tier("utterance")]

      def get_child_tiers_for_tier(self, tier):
            if tier.name == "utterance":
                  return [poioapi.io.graf.Tier("word")]

            return None

      def get_annotations_for_tier(self, tier, annotation_parent=None):
            if tier.name == "utterance":
                  return [poioapi.io.graf.Annotation(i, v) for i, v in
                  enumerate(self.utterance_tier)]

            if tier.name == "word":
                  return [poioapi.io.graf.Annotation(2 + 4
*annotation_parent.id + i, v) for i, v in
enumerate(self.word_tier[annotation_parent.id])]

            return []

      def tier_has_regions(self, tier):
            if tier.name == "utterance":
                  return True

            return False

      def region_for_annotation(self, annotation):
            if annotation.id == 0:
                  return (0, len(self.utterance_tier[0]))
            elif annotation.id == 1:
                  return (len(self.utterance_tier[0]) + 1,
                  len(self.utterance_tier[0]) + 1 +
len(self.utterance_tier[1]))

      def get_primary_data(self):
            return ' '.join(self.utterance_tier)
```

B. Exemplo da implementação de um filtros do módulo annotationgraph

```
ag = poioapi.annotationgraph.AnnotationGraph()

ag.from_elan("elan-example.eaf")

# Atribuir a estrutura hierárquica
ag.structure_type_handler =
poioapi.data.DataStructureType(ag.tier_hierarchies[1])

# A hierarquia ag.tier_hierarchies[1] é a seguinte
#     ['utterance..W-Spch',
#           ['words..W-Words',
#                 ['part_of_speech..W-POS']],
#           ['phonetic_transcription..W-IPA']]

af = poioapi.annotationgraph.AnnotationGraphFilter(ag)

af.set_filter_for_tier("words..W-Words", "follow")
af.set_filter_for_tier("part_of_speech..W-POS", r"\bpro\b")

# Aplicar o filtro ao AnnotationGraph
ag.append_filter(af)

# Imprimir o resultado
print(ag.filtered_node_ids)

#[['utterance..W-Spch..na10',
#'utterance..W-Spch..na12',
#'utterance..W-Spch..na19']]
```

C. Exemplo da implementação dos testes aplicados ao parser do formato de dados do software ELAN

```
class TestElan:
      """
      This class contain the test methods to the
      class io.elan.py.
      """

      # Define os atributos comuns para todos os testes
      def setup(self):
            self.filename = "example.eaf"

            self.parser = poioapi.io.elan.Parser(self.filename)

      # Teste relacionados com o Parser
      def test_get_root_tiers(self):
            root_tiers = self.parser.get_root_tiers()

            expected_value = 4

            # Faz a validação do teste
            assert len(root_tiers) == expected_value

      def test_get_child_tiers_for_tier(self):
            # Get the root tiers
            root_tiers = self.parser.get_root_tiers()

            # Select the W-Spch tier
            tier = root_tiers[1]

            child_tier = self.parser.get_child_tiers_for_tier(tier)

            expected_value = 2

            # Faz a validação do teste
            assert len(child_tier) == expected_value

      [ … ]
```

D. Legenda dos símbolos utilizados nos fluxogramas